语言模型

语言模型通俗的将就是判断一句话是不是正常人说出来的。统计语言模型是所有 NLP 的基础，被广泛应用与语音识别、机器翻译、分词、词性标注和信息检索等任务。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，这个概率分布也是该语言的生成模型。通俗的讲，如果一句话没有在语料库中出现，可以模拟句子的生成的方式，生成句子在语料库中的概率。一般语言模型可以使用各个词语条件概率的形式表示：

$$p(w_1^n) = p(w_1, w_2, \dots, w_n) = \prod_{i=1}^{n} p(w_i \mid Context)$$

其中，Context 为 w_i 的上下文表示。根据 Context 的表示差异，统计语言模型又可以分为不同的类别，其中最具代表性的有 n-gram 语言模型及 nn 语言模型:

N-gram

是自然语言处理（NLP）中一个非常重要的概念，通常在 NLP 中，人们基于一定的语料库，可以利用 N-gram 来做以下几类事情：

1. 预计或者评估一个句子是否合理；
2. 评估两个字符串之间的差异程度，这也是模糊匹配中常用的一种手段；
3. 语音识别；
4. 机器翻译；
5. 文本分类。

概率模型

统计语言模型实际上是一个概率模型，所以常见的概率模型都可以用于求解这些参数
常见的概率模型有：N-gram 模型、决策树、最大熵模型、隐马尔可夫模型、条件随机场、神经网络等
目前常用于语言模型的是 N-gram 模型和神经语言模型

我 今天 下午 打 篮球
p(S)=p(w1,w2,w3,w4,w5,...,wn)=p(w1)p(w2|w1)p(w3|w1,w2)...p(wn|w1,w2,...,wn-1)

p(S)被称为语言模型，即用来计算一个句子概率的模型

条件概率公式：

P(AB)=P(A|B)*P(B)

P(A|B)=P(AB)/P(B)

p(wi|w1,w2,...,wi-1)=p(w1,w2,...,wi-1,wi)/p(w1,w2,...,wi-1)

由于要计算 wi 出现的概率，就要去统计前 i-1 词出现的情况，假设词库中有 n 个词，就有 n^(i-1)种可能，这样每增加一个单词，模型的计算成本都指数倍的增长。

- 数据过于稀疏
- 参数空间太大

马尔可夫(Markov)假设——未来的事件，只取决于有限的历史

基于马尔可夫假设，N-gram 语言模型认为一个词出现的概率只与它前面的 n-1 个词相关

假设下一个词的出现依赖它前面的一个词：

p(S)=p(w1,w2,w3,w4,w5,...,wn)=p(w1)p(w2|w1)p(w3|w1,w2)...p(wn|w1,w2,...,wn-1)

=p(w1)p(w2|w1)p(w3|w2)...p(wn|wn-1)

大数定律：

$$p(w_k|w_{k-1}) = \frac{p(w_{k-1}, w_k)}{p(w_{k-1})} \approx \frac{\text{count}(w_{k-1}, w_k)}{\text{count}(w_{k-1})}$$

假设下一个词的出现依赖它前面的两个词：

p(S)=p(w1,w2,w3,w4,w5,...,wn)=p(w1)p(w2|w1)p(w3|w1,w2)...p(wn|w1,w2,...,wn-1)

=p(w1)p(w2|w1)p(w3|w1,w2)...p(wn|wn-1,wn-2)

通常情况下，我们管这样的叫 n-gram 模型

当 N=1 的时候，为一元模型（Unigram model）：

P(S)=P(w1,w2,w3,..,wn)=P(W1)*P(W2)*...*P(Wn)

当 N=2 的时候，叫二元模型（Bigram model）：

P(S)=P(w1,w2,w3,..,wn)=P(W1|start)*P(W2|W1)*...*P(Wn|Wn-1)

当 N=3 的时候，叫三元模型（trigram model）：

P(S)=P(w1,w2,w3,..,wn)=P(W2|start,W1)*P(W3|W1,W2)*...*P(Wn|Wn-1,Wn-2)

也许你已经意识到，N 是一个超参数，在面临实际问题时，我们应该如何选择依赖词的个数？

更大的 n：对下一个词出现的约束信息更多，具有更大的辨别力；

更小的 n：在训练语料库中出现的次数更多，具有更可靠的统计信息，具有更高的可靠性。

理论上，n 越大越好，但在经验上看，trigram 用的最多，尽管如此，原则上，能用 bigram 解决，绝不使用 trigram。

i	want	to	eat	chinese	food	lunch	spend
2533	927	2417	746	158	1093	341	278

二元单词分布情况表

	i	want	to	eat	chinese	food	lunch	spend
i	5	827	0	9	0	0	0	2
want	2	0	608	1	6	6	5	1
to	2	0	4	686	2	0	6	211
eat	0	0	2	0	16	2	42	0
chinese	1	0	0	0	0	82	1	0
food	15	0	15	0	1	4	0	0
lunch	2	0	0	0	0	1	0	0
spend	1	0	1	0	0	0	0	0

第一行，第二列表示给定前一个词是“i”时，当前词为“want”的情况一共出现了 827 次。

频率分布表

	i	want	to	eat	chinese	food	lunch	spend
i	0.002	0.33	0	0.0036	0	0	0	0.00079
want	0.0022	0	0.66	0.0011	0.0065	0.0065	0.0054	0.0011
to	0.00083	0	0.0017	0.28	0.00083	0	0.0025	0.087
eat	0	0	0.0027	0	0.021	0.0027	0.056	0
chinese	0.0063	0	0	0	0	0.52	0.0063	0
food	0.014	0	0.014	0	0.00092	0.0037	0	0
lunch	0.0059	0	0	0	0	0.0029	0	0
spend	0.0036	0	0.0036	0	0	0	0	0

因为我们从表 1 中知道“i”一共出现了 2533 次，而其后出现“want”的情况一共有 827 次，所以 P(want|i)=827/2533≈0.33。

I want chinese food

p(I want chinese food)=p(want|I)*p(chinese|want)*p(food|chinese)

可靠性与可区别性

假设没有计算和存储限制，n 是不是越大越好？

早期因为计算性能的限制，一般最大取到 n=4；如今，即使 n>10 也没有问题，

但是，随着 n 的增大，模型的性能增大却不显著，这里涉及了可靠性与可区别性的问题

参数越多，模型的可区别性越好，但是可靠性却在下降——因为语料的规模是有限的，导致 count(W) 的实例数量不够，从而降低了可靠性

OOV 问题

OOV 即 Out Of Vocabulary，也就是序列中出现了词表外词，或称为未登录词

或者说在测试集和验证集上出现了训练集中没有过的词

一般解决方案：

**设置一个词频阈值，只有高于该阈值的词才会加入词表

**所有低于阈值的词替换为 UNK（一个特殊符号）

无论是统计语言模型还是神经语言模型都是类似的处理方式

平滑处理

count(W) = 0 是怎么办？

平滑方法：

Add-one Smoothing (Laplace)

总结

该语言模型只看当前词的前 n-1 个词对当前词的影响。所有该模型的优势为：（1）该模型包含了前 n-1 个词所能提供的全部信息，这些词对当前词的出现具有很强的约束能；（2）只看前 n-1 个词而非所有词，所以使得模型的效率较高。该模型也有很多缺陷：（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（通常 n=2 或 3）；（2）该模型无法建模出词与词之间的相似度，比如“白色汽车”和“白色轿车”；（3）训练语料中有些 n 元组没有出现过，其对应的条件概率为 0，导致计算一整句话的概率为 0！

神经网络语言模型(NPLM)

$$p(w|\text{context}(w)) = g(i_w, V_{context})$$

1. 其中 g 表示神经网络，i_w 为 w 在词表中的序号，context(w) 为 w 的上下文，V_context 为上下文构成的特征向量。
2. V_context 由上下文的词向量进一步组合而成

N-gram 神经语言模型

A Neural Probabilistic Language Model (Bengio, et al., 2003)

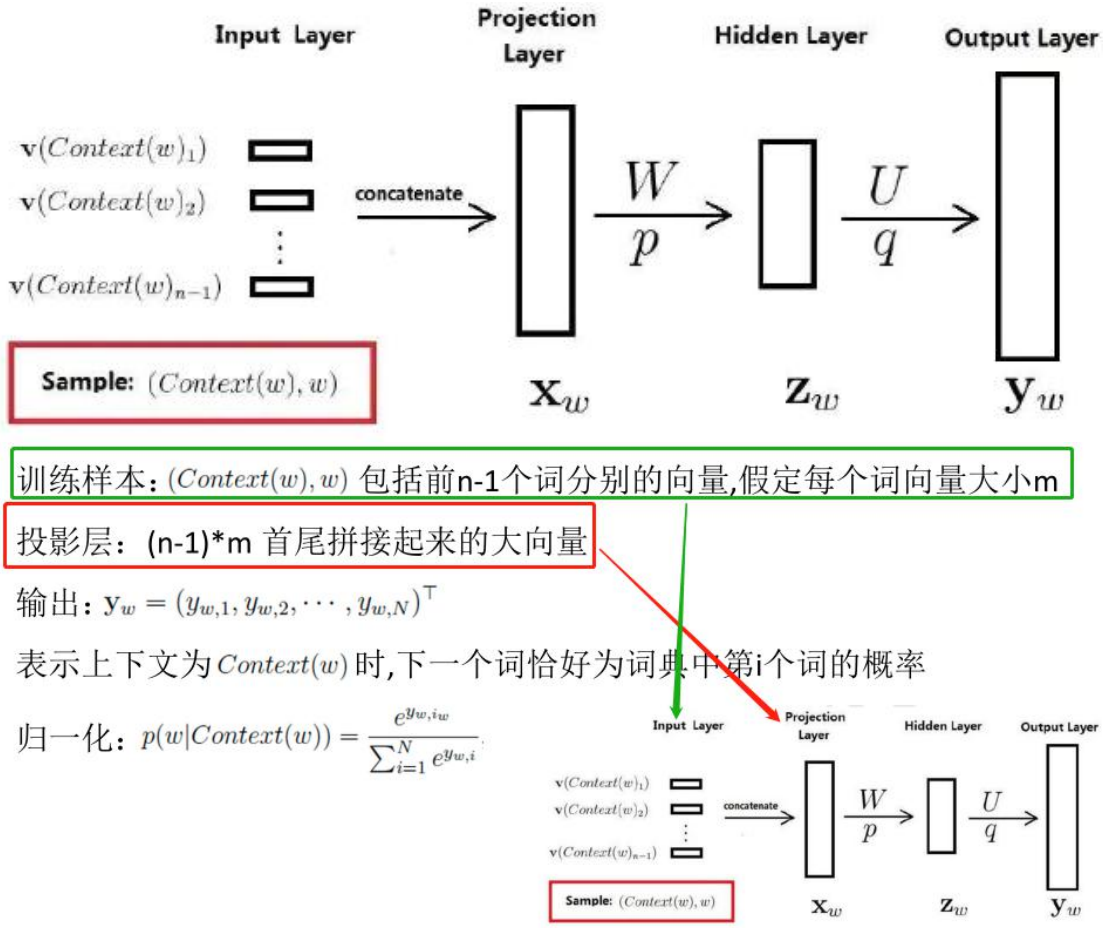
这是一个经典的神经概率语言模型，它沿用了 N-gram 模型中的思路，将 w 的前 n-1 个词作为 w 的上下文 context(w)，而 V_context 由这 n-1 个词的词向量拼接而成，即

$$p(w_k|w_{k-n+1}^{k-1}) = g(i_{w_k}, [c(w_{k-n+1}); \ldots; c(w_{k-1})])$$

1. 其中 c(w) 表示 w 的词向量

2. 不同的神经语言模型中 context(w) 可能不同，比如 Word2Vec 中的 CBOW 模型
3. 每个训练样本是形如 (context(w), w) 的二元对，其中 context(w) 取 w 的前 n-1 个词；当不足 n-1，用特殊符号填充
4. 同一个网络只能训练特定的 n，不同的 n 需要训练不同的神经网络

N-gram 神经语言模型的网络结构



训练样本：$(Context(w), w)$ 包括前n-1个词分别的向量,假定每个词向量大小m

投影层：(n-1)*m 首尾拼接起来的大向量

输出：$\mathbf{y}_w = (y_{w,1}, y_{w,2}, \cdots, y_{w,N})^\top$

表示上下文为 $Context(w)$ 时,下一个词恰好为词典中第i个词的概率

归一化：$p(w|Context(w)) = \frac{e^{y_{w,i_w}}}{\sum_{i=1}^{N} e^{y_{w,i}}}$

- ✓ 【输入层】首先，将 context(w) 中的每个词映射为一个长为 m 的词向量，词向量在训练开始时是随机的，并参与训练；
- ✓ 【投影层】将所有上下文词向量拼接为一个长向量，作为 w 的特征向量，该向量的维度为 m(n-1)
- ✓ 【隐藏层】拼接后的向量会经过一个规模为 h 隐藏层，该隐层使用的激活函数为 tanh
- ✓ 【输出层】最后会经过一个规模为 N 的 Softmax 输出层，从而得到词表中每个词作为下一个词的概率分布
- ✓ 其中 m, n, h 为超参数，N 为词表大小，视训练集规模而定，也可以人为设置阈值
- ✓ 训练时，使用交叉熵作为损失函数
- ✓ 当训练完成时，就得到了 N-gram 神经语言模型，以及副产品词向量

✓ 整个模型可以概括为如下公式：

$$y = U \cdot \tanh(Wx + p) + q$$

模型参数的规模与运算量

模型的超参数：m, n, h, N

m 为词向量的维度，通常在 10^1 ~ 10^2

n 为 n-gram 的规模，一般小于 5

h 为隐藏的单元数，一般在 10^2

N 位词表的数量，一般在 10^4 ~ 10^5，甚至 10^6

网络参数包括两部分

词向量 C: 一个 N * m 的矩阵——其中 N 为词表大小，m 为词向量的维度

网络参数 W, U, p, q：

```
- W: h * m(n-1) 的矩阵
- p: h * 1      的矩阵
- U: N * h    的矩阵
- q: N * 1    的矩阵
```

模型的运算量

主要集中在隐藏层和输出层的矩阵运算以及 SoftMax 的归一化计算

此后的相关研究中，主要是针对这一部分进行优化，其中就包括 Word2Vec 的工作

相比 N-gram 模型，NPLM 的优势

单词之间的相似性可以通过词向量来体现

S1=“我 今天 去 网咖” 出现了 1000 次

S2=“我 今天 去 网吧” 出现了 10 次

对于 N-gram 模型：P(S1)>>P(S2)

而神经网络模型计算的 P(S1)≈P(S2)

Word2Vec

One-hot representation：对应位置数字取 1，其他位置全为 0

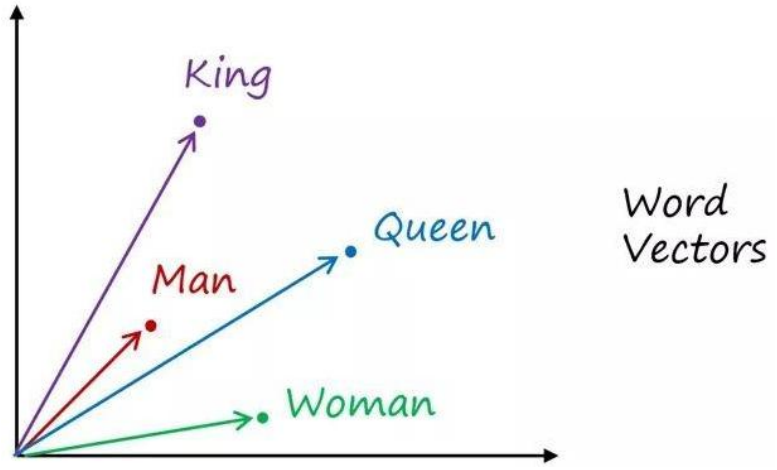
$$\begin{pmatrix} the \\ cat \\ sat \\ on \\ the \\ mat \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

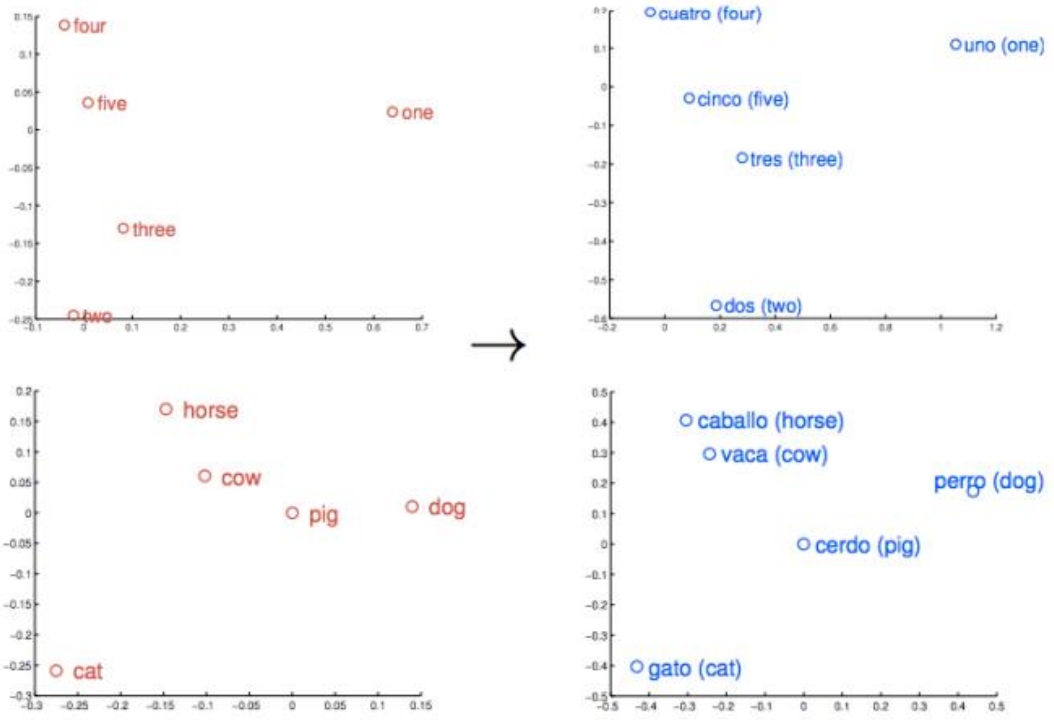
计算机是无法直接处理文本信息的，所以，在我们构建神经网络之前，要对文本进行一定的处理。

相信大家对独热编码（one-hot encode）应该不陌生了，虽说它能把所有文本用数字表示出来，但是表示文本的矩阵会非常的稀疏，极大得浪费了空间，而且这样一个矩阵放入神经网络训练也会耗费相当多的时间。

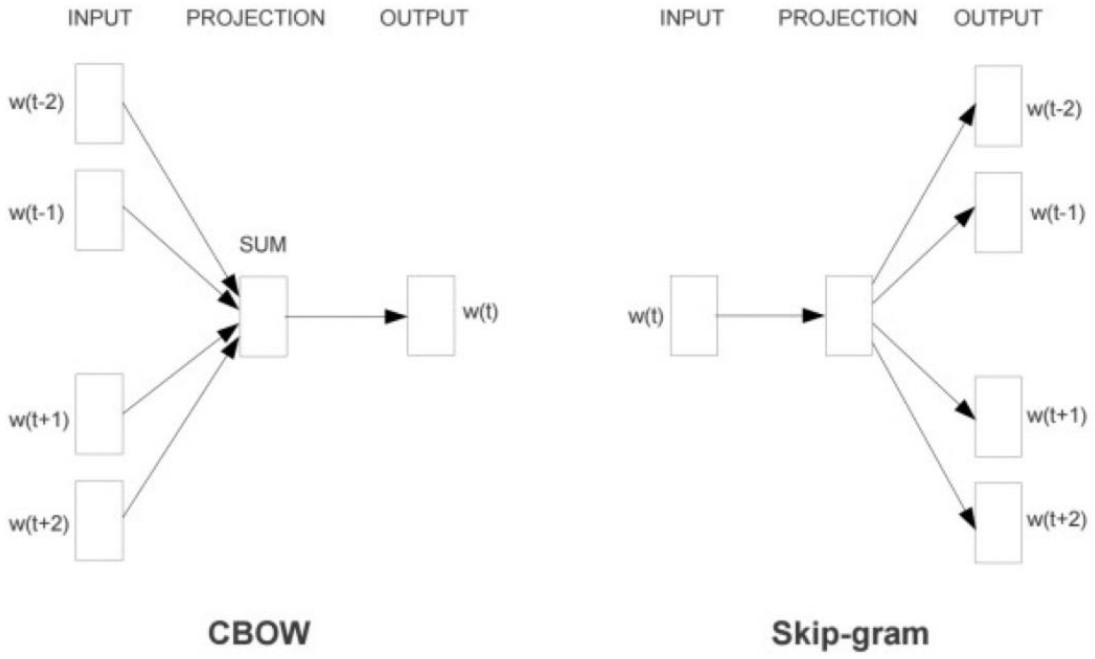
为此，提出了词向量模型(Word2Vec)。词向量模型是一种将词的语义映射到向量空间的技术，说白了就是用向量来表示词，但是会比用独热编码用的空间小，而且词与词之间可以通过计算余弦相似度来看两个词的语义是否相近，显然 King 和 Man 两个单词语义更加接近，而且通过实验我们知道 King-Man+Woman=Queen

Distributed representation：通过训练将每一个词都映射为一个 K 维的浮点数向量



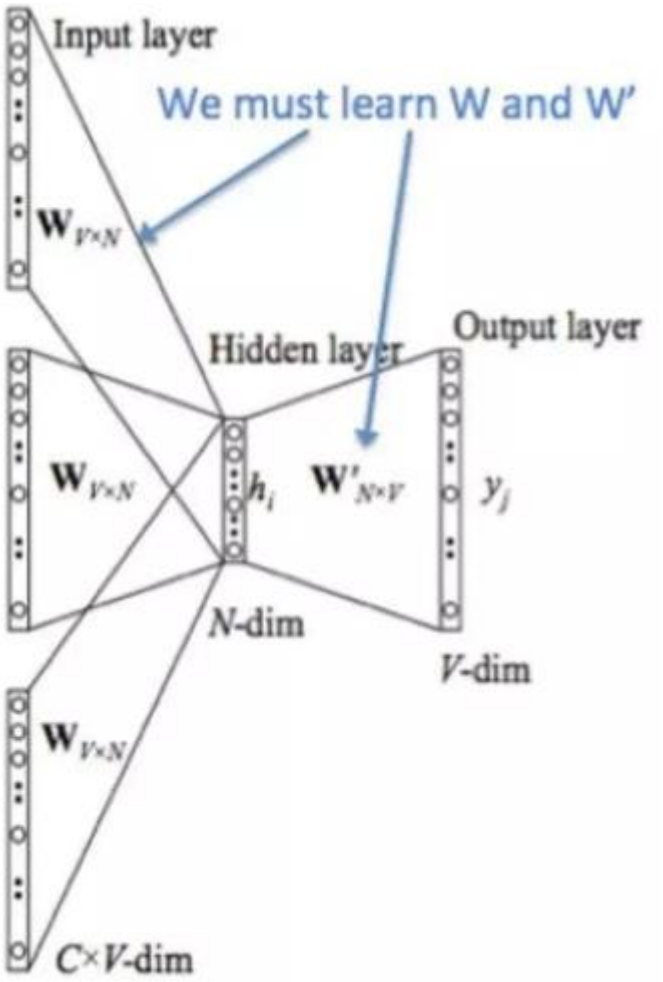


有了前面的准备，本节正式介绍 word2vec 中用到的两个重要的模型-CBOW 模型（Continuous Bag-of-Words Model）和 Skip-gram 模型（Continuous Skip-gram Model）。



CBOW

根据上下文预测出来当前这个词是什么



需要注意的是：对于任意的单词，Input layer 和 Hidden Layer 之间的权重矩阵 W 是参数共享的

1. 输入层：上下文单词的 onehot. {假设单词向量空间 dim 为 V，上下文单词个数为 C}
2. 所有 onehot 分别乘以共享的输入权重矩阵 W. {V*N 矩阵，N 为自己设定的数，初始化权重矩阵 W}
3. 所得的向量 {因为是 onehot 所以为向量} 相加求平均作为隐层向量, size 为 1*N.
4. 乘以输出权重矩阵 W' {N*V}
5. 得到向量 {1*V} 激活函数处理得到 V-dim 概率分布 {PS: 因为是 onehot 嘛，其中的每一维斗代表着一个单词} ,概率最大的 index 所指示的单词为预测出的中间词(target word)
6. 与 true label 的 onehot 做比较，误差越小越好

所以，需要定义 loss function（一般为交叉熵代价函数），采用梯度下降算法更新 W 和 W'。训练完毕后，输入层的每个单词与矩阵 W 相乘得到的向量的就是我们想要的词向量（word embedding），这个矩阵（所有单词的 word embedding）也叫做 look up table（其实聪明的你已经看出来了，其实这个 look up table 就是矩阵 W 自身），也就是说，任何一个单词的 onehot 乘以这个矩阵都将得到自己的词向量。有了 look up table 就可以

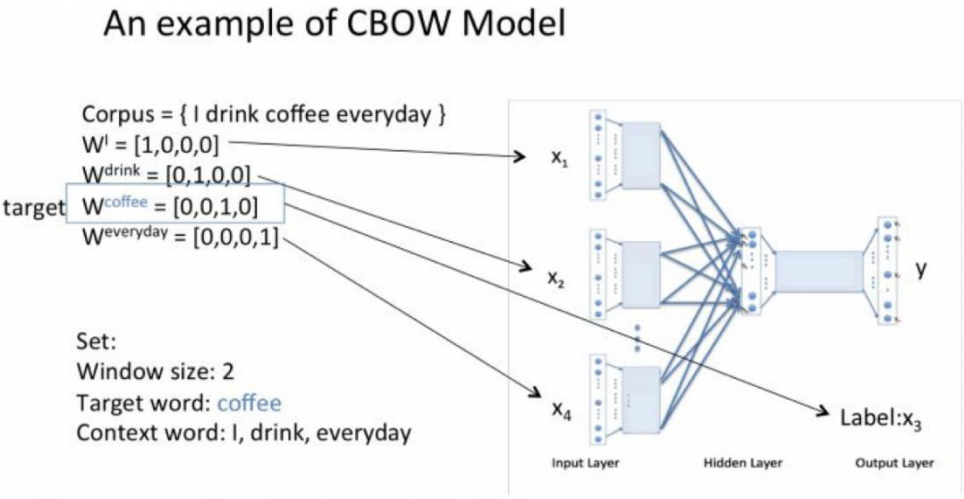
免去训练过程直接查表得到单词的词向量了。

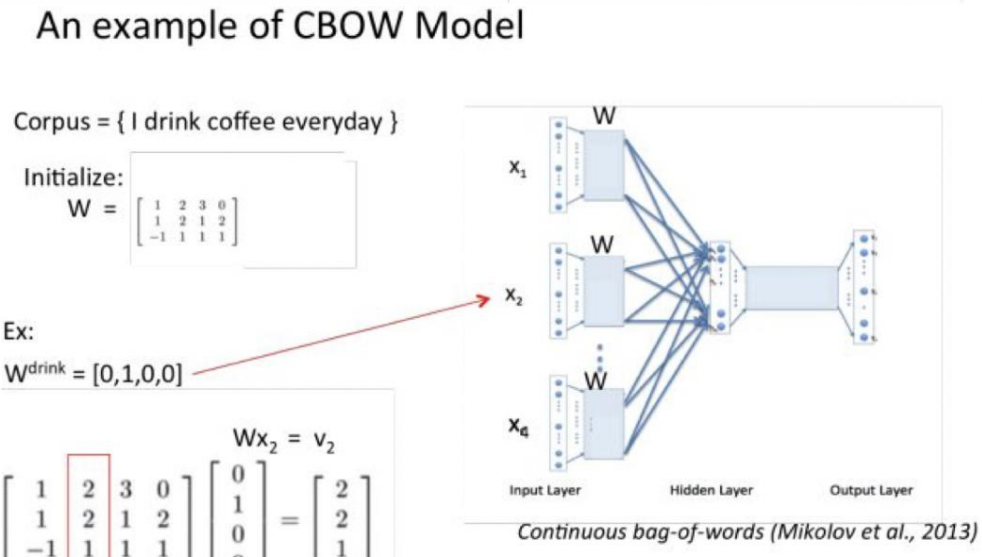
假设我们现在的 Corpus 是这一个简单的只有四个单词的 document：

{I drink coffee everyday}

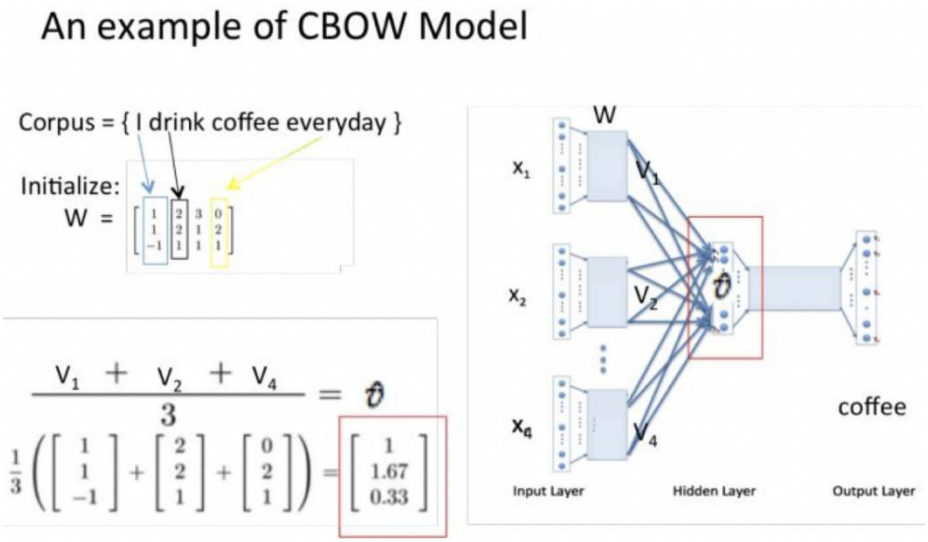
我们选 coffee 作为中心词，window size 设为 2

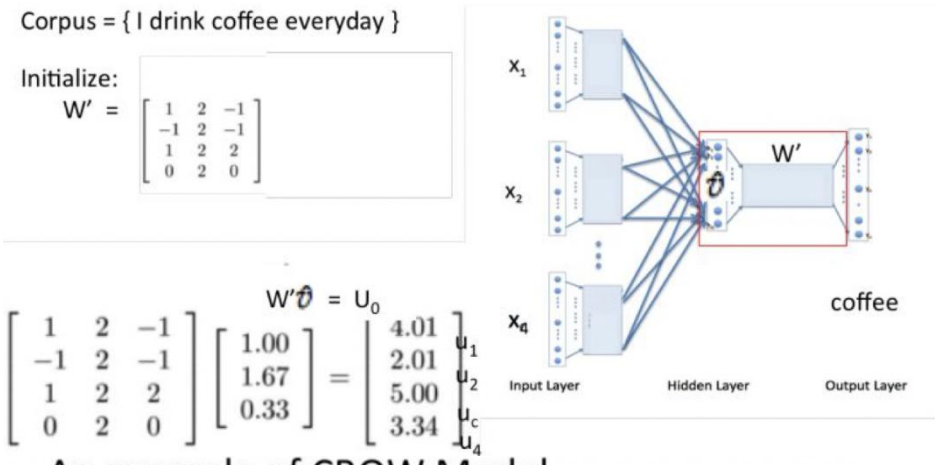
也就是说，我们要根据单词"I","drink"和"everyday"来预测一个单词，并且我们希望这个单词是 coffee。

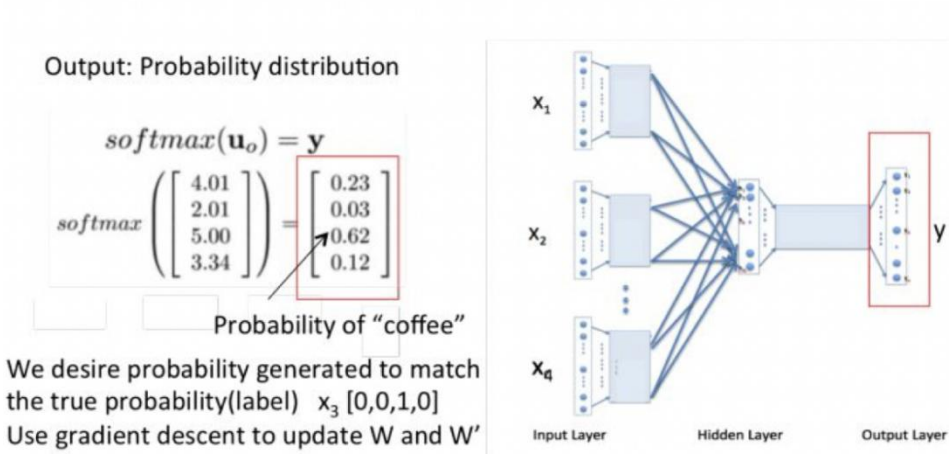




Continuous bag-of-words (Mikolov et al., 2013)







假设我们此时得到的概率分布已经达到了设定的迭代次数，那么现在我们训练出来的 look up table 应该为矩阵 W。即，任何一个单词的 one-hot 表示乘以这个矩阵都将得到自己的 word embedding。

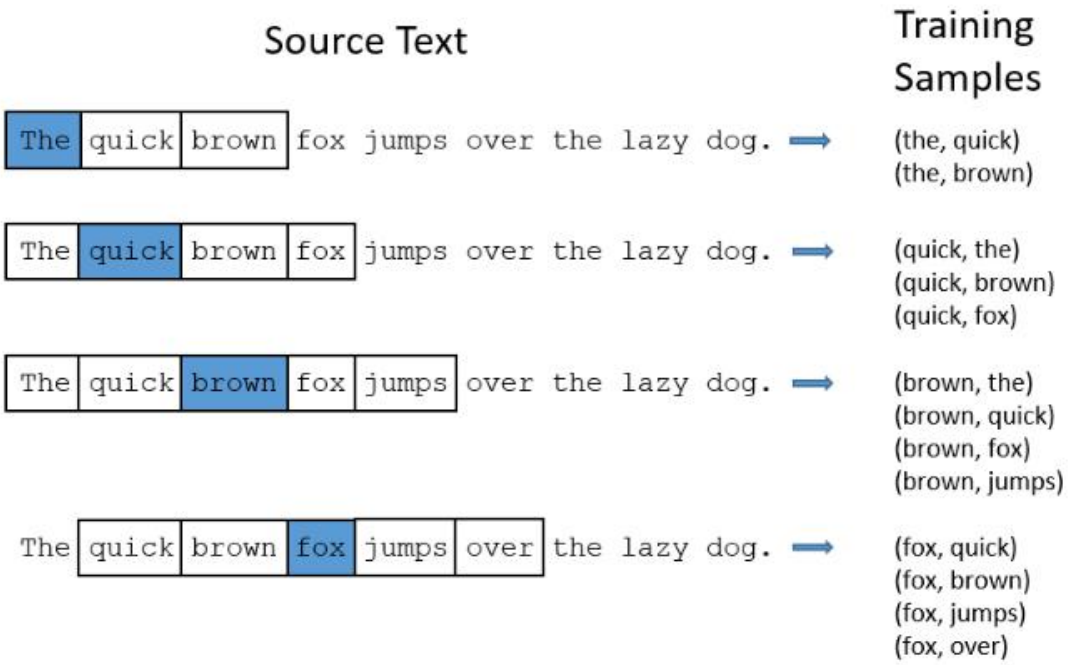
Continuous Bag-of-Word Model(连续词袋模型)和 skip-gram model(跳字模型)，

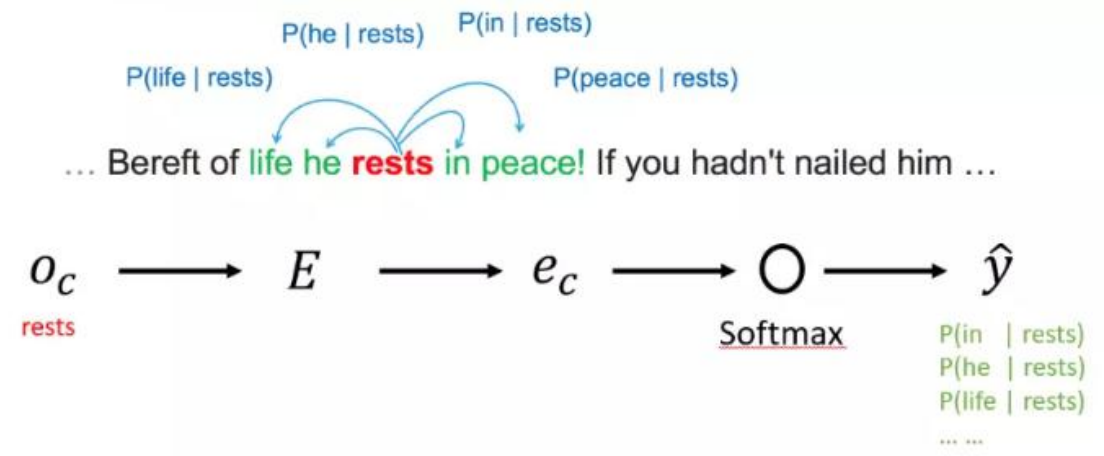
分别对应了词向量的两种训练方法：利用 context 预测中心词以及利用中心词去预测 context。对于连续词袋模型（CBOW）来说，一般的做法是先对每个单词进行 one-of-N 编码（one-hot encoded），作为训练网络的输入，接着构建一层 hidden layer，最后构建输出层，这一层是一个 softmax 层，每个 context 单词到中心单词的事件都被认为是独立的，所以将这些事件发生的概率相乘，最后构建损失函数，即：将输出概率分布和实际选中的词概率分布进行 Corss Entropy 计算，接下来使用 SGD 对参数进行更新。这里，hidden layer 的训练结果就是最终的 word vector 了。

Skip-gram

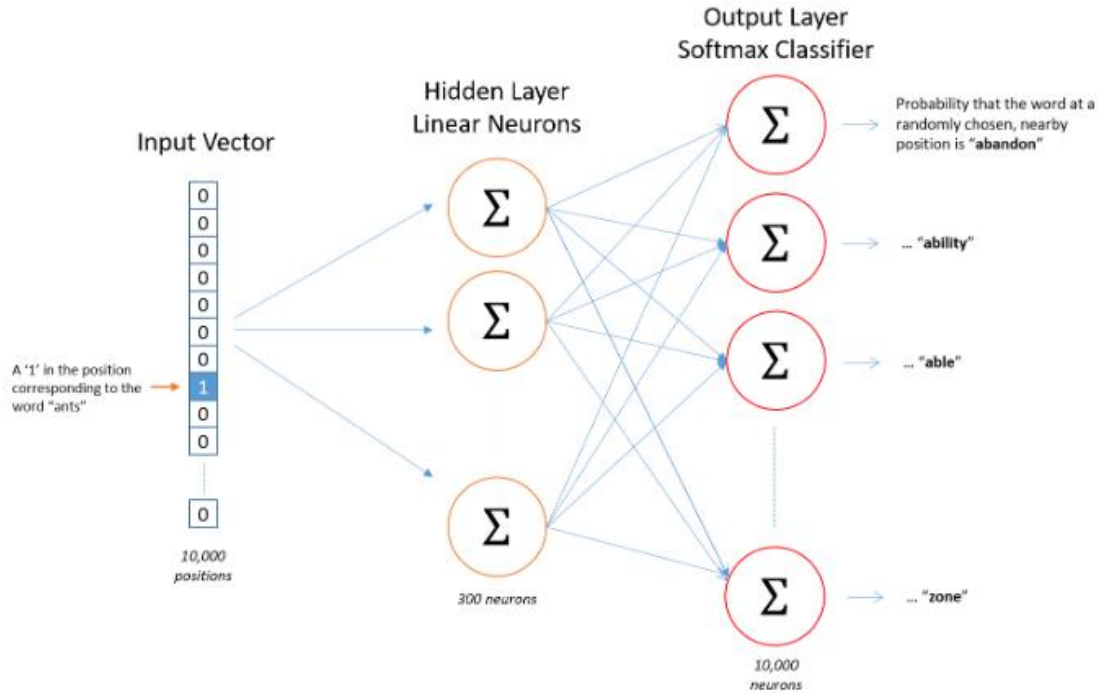
根据当前这个词来预测上下文

Skip-gram 的具体训练过程如下，蓝色代表输入的词，图的框框代表滑动窗口，用来截取蓝色词的上下文，蓝色词的上下文作为输出，然后形成训练标本（Training Samples），这样我们就得到了{输入和输出}，将他们放入{输入层-隐藏层-输出层}的神经网络训练，我们就能得到 Skip-gram 模型。因为神经网络不能直接处理文本，因此所有的词都用 one-hot encode 表示。





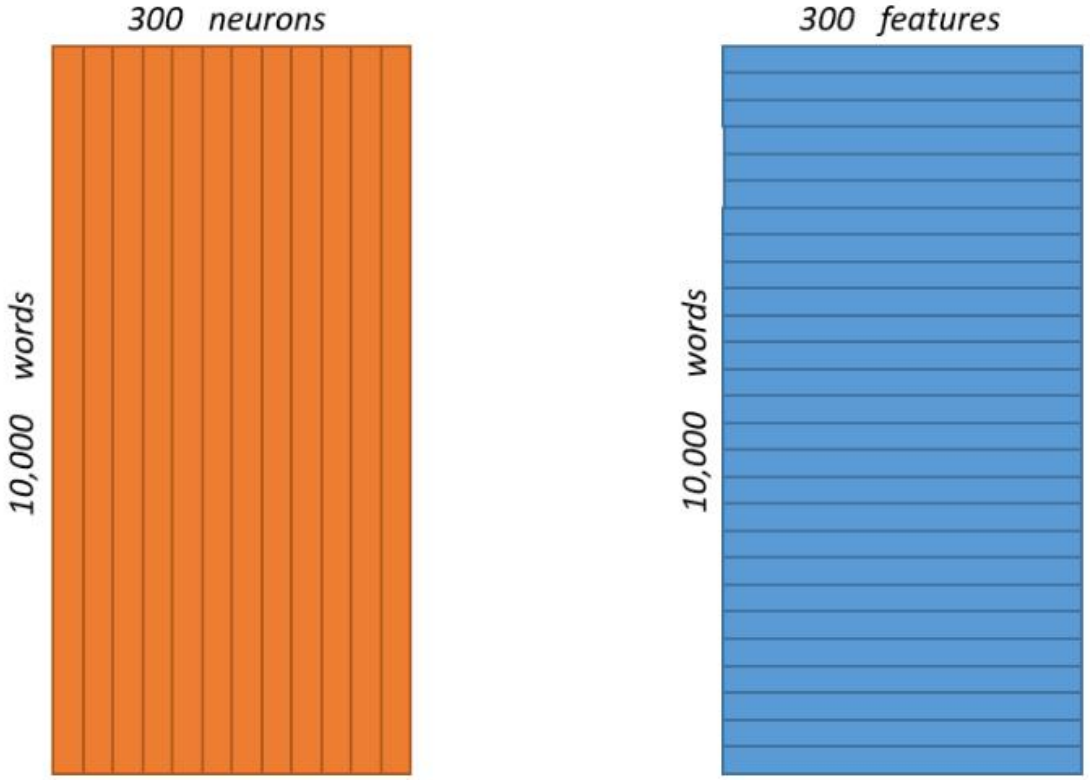
Skip-gram 的神经网络结构如图所示，隐藏层有 300 个神经元，输出层用 softmax 激励函数，通过我们提取的词与其相应的上下文去训练，得到相应的模型。通过 Softmax 激励函数，输出层每个神经元输出的是概率，加起来等于 1。



但输出层并不是我们关心的，我们去掉模型的输出层，才是我们想要的词向量模型，我们通过隐藏层的权重来表示我们的词。

现在假设我们有 10000 个词，每个词用 one-hot encode 编码，每个词大小就是 1*10000，现在我们想用 300 个特征去表示一个词，那么隐藏层的输入是 10000，输出是 300（即 300 个神经元），因此它的权值矩阵大小为 10000 * 300。那么我们的词向量模型本质上就变成了矩阵相乘。

Hidden Layer Weight Matrix → *Word Vector Lookup Table!*



$$[0 \quad 0 \quad 0 \quad 1 \quad 0] \times \begin{bmatrix} 17 & 24 & 1 \\ 23 & 5 & 7 \\ 4 & 6 & 13 \\ 10 & 12 & 19 \\ 11 & 18 & 25 \end{bmatrix} = [10 \quad 12 \quad 19]$$

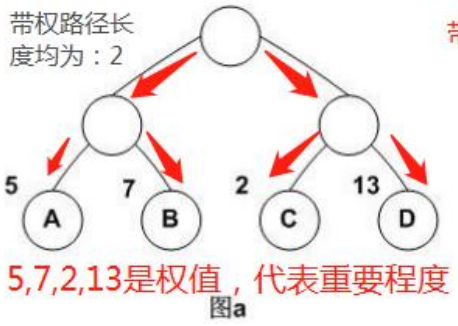
- word2vec
  - 基于层次SoftMax的模型
    - CBOW 模型
    - Skip-gram 模型
  - 基于Negative sampling的模型
    - CBOW 模型
    - Skip-gram 模型

$$\mathcal{L} = \sum_{w \in \mathcal{C}} \log p(w|Context(w)).$$

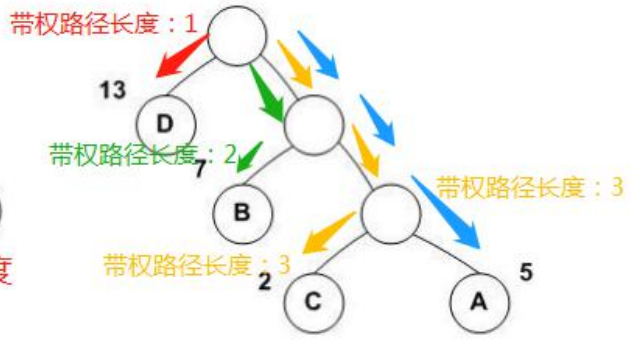
Hierarchical Softmax

哈夫曼树

哈夫曼树是一种带权路径长度最短的二叉树，也称为最优二叉树



图a



图b 哈夫曼树

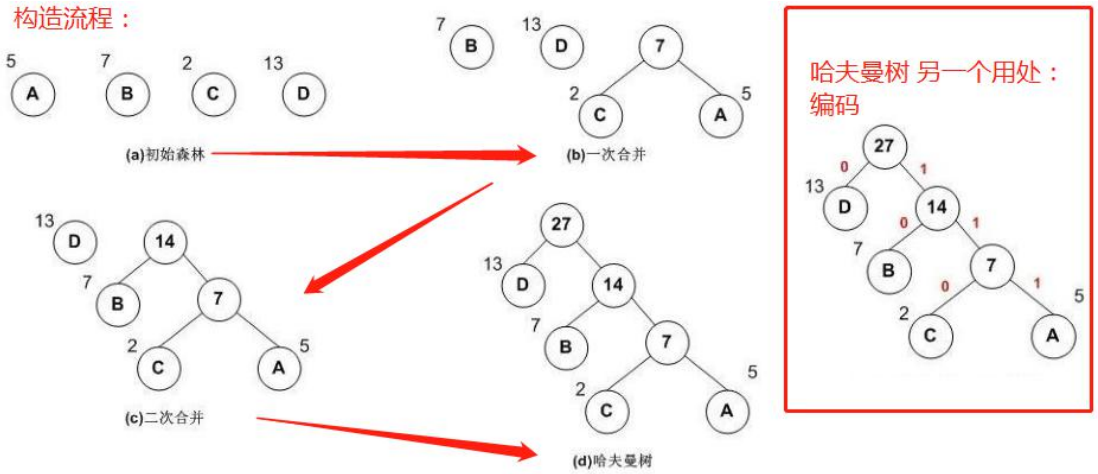
他们的带权路径长度分别为：

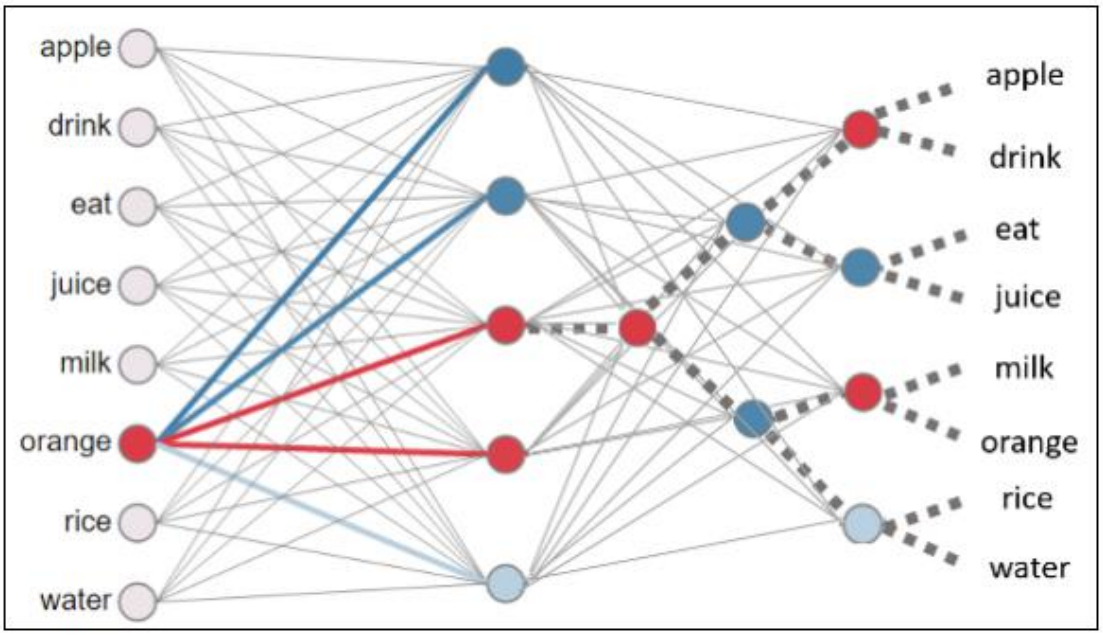
图 a：WPL=5*2+7*2+2*2+13*2=54

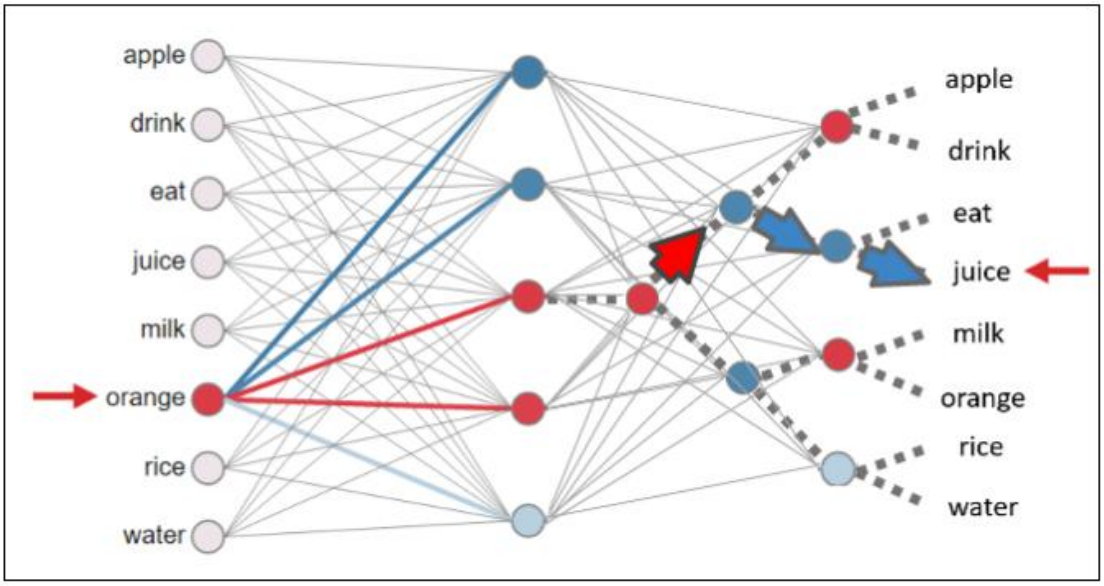
图 b：WPL=5*3+2*3+7*2+13*1=48

所以 b 带权路径长度较小，图 b 就是哈夫曼树，最优二叉树

分层 softmax 就是可以把节点 A B C D 看成是一个个词，权重看成是词出现的次数或频率

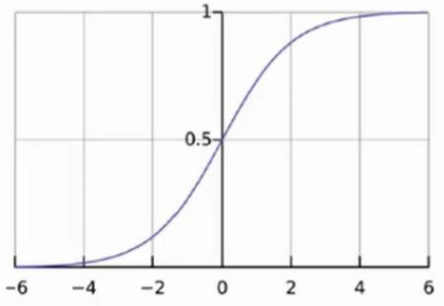






二叉树每一个分裂都是一个条件判断，相当于二分类，sigmoid 函数来判断走左边还是右边

$$h_\theta(x)=g(\theta^T x)=\frac{1}{1+e^{-\theta^T x}}$$



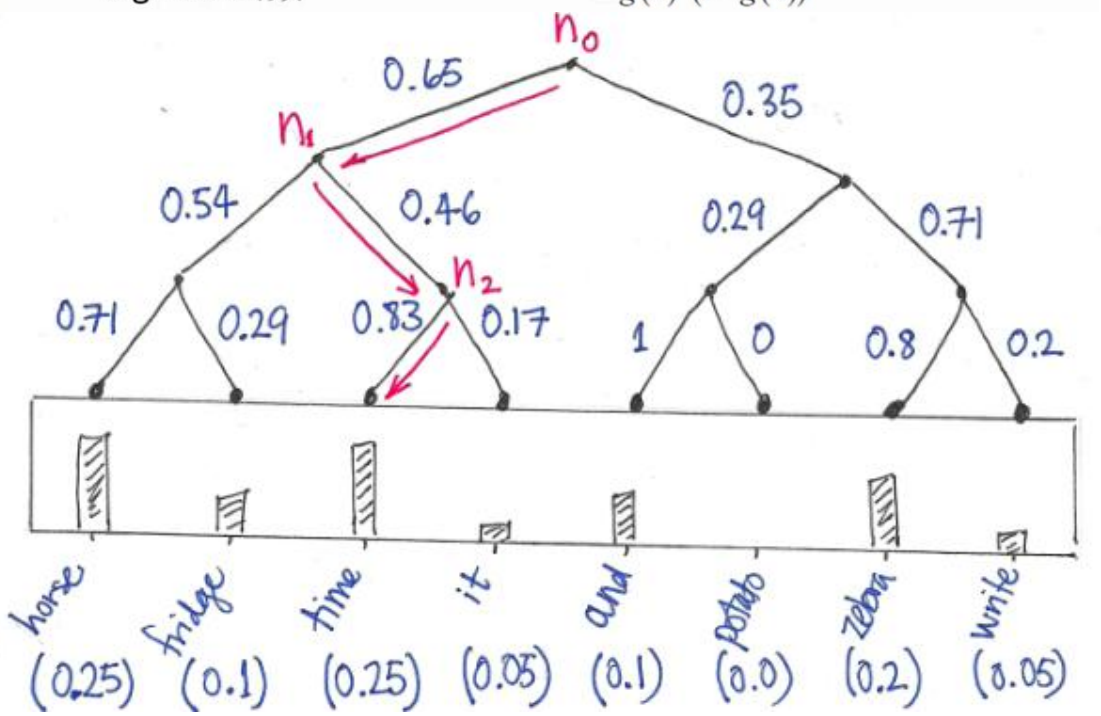
Sigmoid函数

$$g(z)=\frac{1}{1+e^{-z}}$$

$$g'(x)=\left(\frac{1}{1+e^{-x}}\right)=\frac{e^{-x}}{\left(1+e^{-x}\right)^2}$$

$$=\frac{1}{1+e^{-x}}\cdot\frac{e^{-x}}{1+e^{-x}}=\frac{1}{1+e^{-x}}\cdot\left(1-\frac{1}{1+e^{-x}}\right)$$

$$=g(x)\cdot(1-g(x))$$



sigmoid 函数是用来做二分类的，在这里正好合适；当路径上的所有二分类的概率都连乘后，得到的就是预测单词的概率，可以证明，词典中所有单词被预测到的概率和为 1。这也是这个方法被叫做分层 softmax 的原因了。

CBOW

输入层是上下文的词语的词向量 ,在训练 CBOW 模型 ,词向量只是个副产品 ,是 CBOW 模型的一个参数，训练开始的时候，词向量是个随机值，随着训练的进行不断被更新。

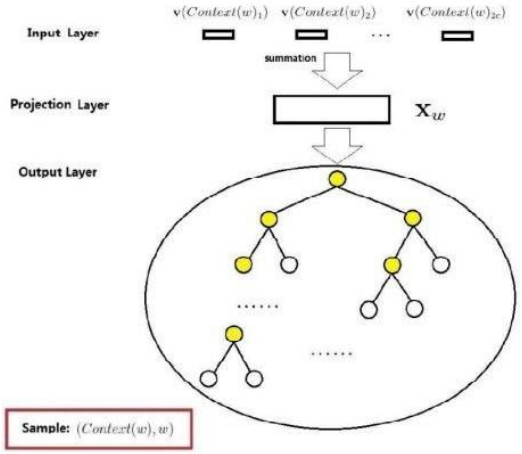
投影层对其求和，就是简单的向量加法。

输出层输出最可能的 word，由于语料库中词汇量是固定的 C 个，所以可以看成是一个多分类问题。

比较 CBOW 模型与 NNLM 的不同：

- ✧ （从输入层到投影层的操作）CBOW-拼接；NNLM-求和累加
- ✧ （隐藏层）CBOW-无耗时的非线性隐藏层； NNLM-非线性隐藏层
- ✧ （输出层）CBOW-树形结构； NNLM-线性结构

NNLM 的大部分计算集中在隐藏层和输出层之间的矩阵向量运算、输出层上的 SoftMax 归一化运算，CBOW 模型对这些计算复杂度高的地方进行了改变：去掉了隐藏层、输出层改用 Huffman 树。



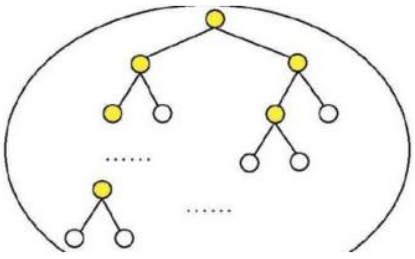
输入层是上下文的词语的词向量，在训练CBOW模型，词向量只是个副产品，确切来说，是CBOW模型的一个参数。训练开始的时候，词向量是个随机值，随着训练的进行不断被更新）。m中的每一个值和权重参数被更新

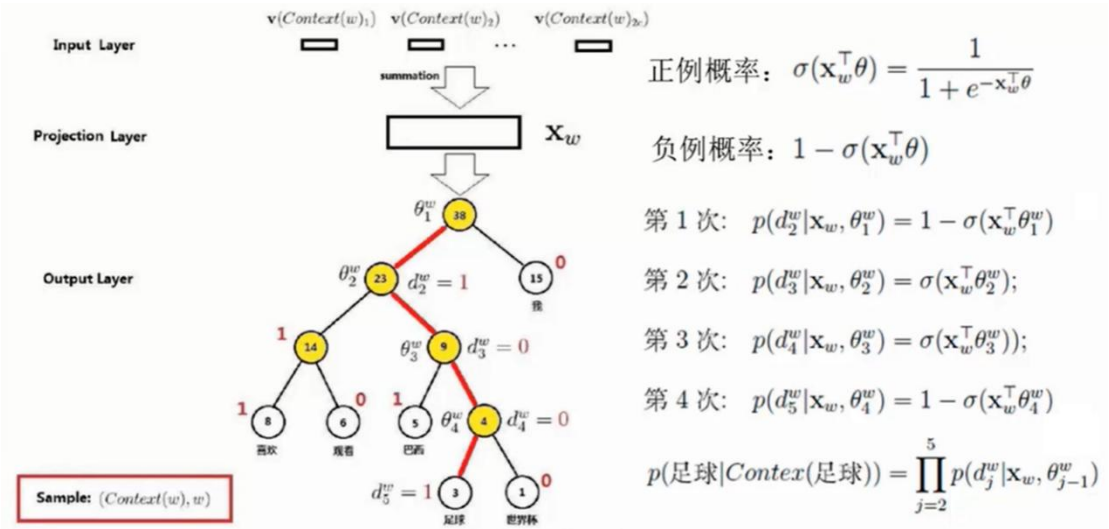
投影层对其求和，所谓求和，就是简单的向量加法。

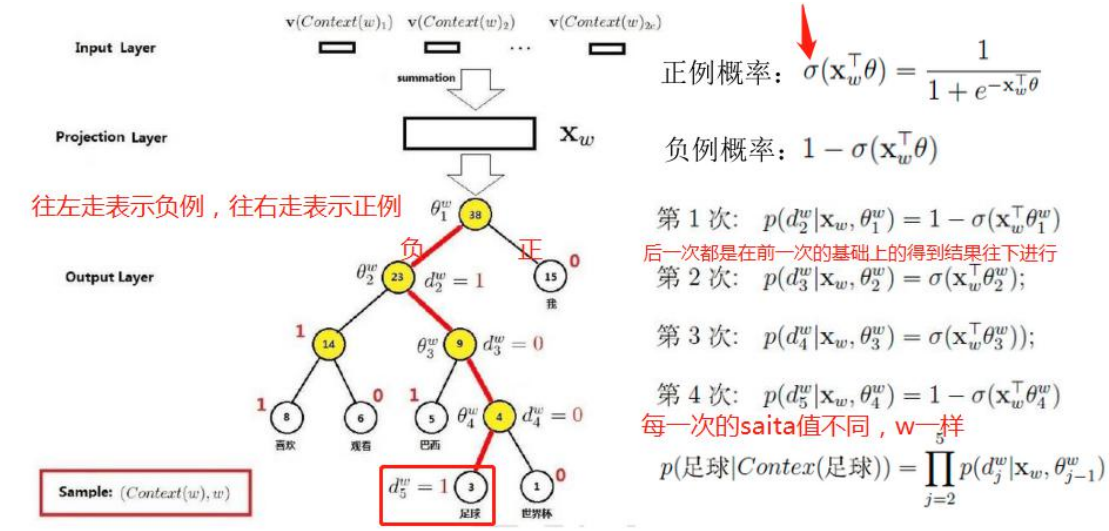
输出层输出最可能的w。由于语料库中词汇量是固定的|C|个，所以上述过程其实可以看做一个多分类问题。给定特征，从|C|个分类中挑一个。

1. p^w从根结点出发到达w对应叶子结点的路径.
2. l^w路径中包含结点的个数
3. $p_1^w, p_2^w, \cdots, p_{l^w}^w$路径 p^w中的各个节点
4. $d_2^w, d_3^w, \cdots, d_{l^w}^w \in \{0,1\}$词w的编码， d_j^w表示路径 p^w第j个节点对应的编码（根节点无编码）
5. $\theta_1^w, \theta_2^w, \cdots, \theta_{l^w-1}^w \in \mathbb{R}^m$路径 p^w中非叶节点对应的参数向量

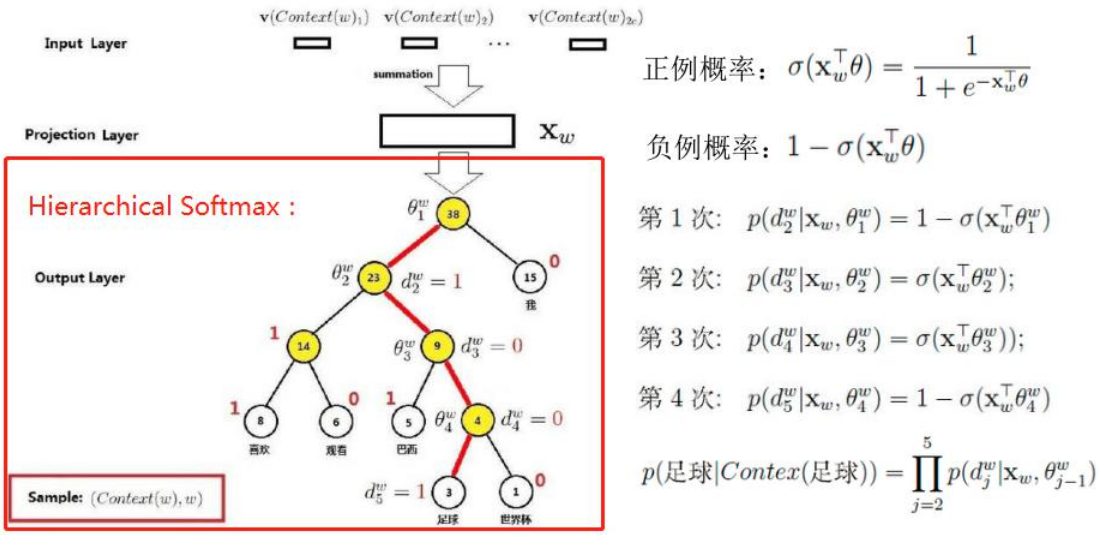
逻辑回归中的输入中的saita







Hierarchical Softmax是用哈夫曼树构造出很多个二分类。



故，基于层次 SoftMax 框架的 CBOW 模型的目标函数(对数似然)可以表示为 L

$$p(d_j^w|\mathbf{x}_w,\theta_{j-1}^w)=[\sigma(\mathbf{x}_w^\top\theta_{j-1}^w)]^{1-d_j^w}\cdot[1-\sigma(\mathbf{x}_w^\top\theta_{j-1}^w)]^{d_j^w}$$

目标函数 → 代入

$$\mathcal{L}=\sum_{w\in\mathcal{C}}\log p(w|Context(w))$$

展开

$$\begin{aligned}\mathcal{L}&=\sum_{w\in\mathcal{C}}\log\prod_{j=2}^{l^w}\{[\sigma(\mathbf{x}_w^\top\theta_{j-1}^w)]^{1-d_j^w}\cdot[1-\sigma(\mathbf{x}_w^\top\theta_{j-1}^w)]^{d_j^w}\}\\&=\sum_{w\in\mathcal{C}}\sum_{j=2}^{l^w}\{(1-d_j^w)\cdot\log[\sigma(\mathbf{x}_w^\top\theta_{j-1}^w)]+d_j^w\cdot\log[1-\sigma(\mathbf{x}_w^\top\theta_{j-1}^w)]\}\end{aligned}$$

Word2vec 中采用的优化方法是随机梯度上升法。随机梯度上升算法的做法是：每次取一个样本（context(w), w），对目标函数中的所有（相关）参数做一次更新，该目标函数中需要更新的参数有两个

X_w^T和θ_{j-1}^w

$$L(w,j)=(1-d_j^w)\cdot log[\sigma(X_w^T\theta_{j-1}^w)]+d_j^w\cdot log\,[1-\sigma(X_w^T\theta_{j-1}^w)]$$

$$\frac{\partial\mathcal{L}(w,j)}{\partial\theta_{j-1}^w}=\frac{\partial}{\partial\theta_{j-1}^w}\{(1-d_j^w)\cdot\log[\sigma(\mathbf{x}_w^\top\theta_{j-1}^w)]+d_j^w\cdot\log[1-\sigma(\mathbf{x}_w^\top\theta_{j-1}^w)]\}$$

sigmoid函数的导数：$\sigma'(x)=\sigma(x)[1-\sigma(x)]$ → 求导之后 ← （xA）'=x的转置

代入上上式得到：$(1-d_j^w)[1-\sigma(\mathbf{x}_w^\top\theta_{j-1}^w)]\mathbf{x}_w-d_j^w\sigma(\mathbf{x}_w^\top\theta_{j-1}^w)\mathbf{x}_w$

合并同类项得到：$[1-d_j^w-\sigma(\mathbf{x}_w^\top\theta_{j-1}^w)]\,\mathbf{x}_w$

θ_{j-1}^w的更新表达式 $\theta_{j-1}^w:=\theta_{j-1}^w+\eta\,[1-d_j^w-\sigma(\mathbf{x}_w^\top\theta_{j-1}^w)]\,\mathbf{x}_w$

学习率（步长）　梯度方向　梯度上升，求最大值，用加号

也要更新上下文的词向量（一开始是随机的词向量）：

$$\frac{\partial\mathcal{L}(w,j)}{\partial\mathbf{x}_w}=[1-d_j^w-\sigma(\mathbf{x}_w^\top\theta_{j-1}^w)]\,\theta_{j-1}^w$$

故，参数的更新如下：

$$\theta_{j-1}^w:=\theta_{j-1}^w+\eta\,[1-d_j^w-\sigma(\mathbf{x}_w^\top\theta_{j-1}^w)]\,\mathbf{x}_w$$

有 CBOW 的图示可知

X_w为取上下文$\tilde{w}$求和累加

故根据链式法则有：

$$\frac{\partial L(w,j)}{\partial \tilde{w}} = \frac{\partial L(w,j)}{\partial X_w}\frac{\partial X_w}{\partial \tilde{w}} = \frac{\partial L(w,j)}{\partial X_w} * 1 = \frac{\partial L(w,j)}{\partial X_w}$$

所以，

$$\mathbf{v}(\tilde{w}) := \mathbf{v}(\tilde{w}) + \eta \sum_{j=2}^{l^w} \frac{\partial \mathcal{L}(w,j)}{\partial \mathbf{x}_w}, \quad \tilde{w} \in Context(w)$$

Skip-gram 模型与 CBOW 模型的推导过程大同小异，这里不打算介绍，本节我们只是介绍一下 skip-gram 模型的特点。基于层次 SoftMax 的 skip-gram 模型的网络结构如下：

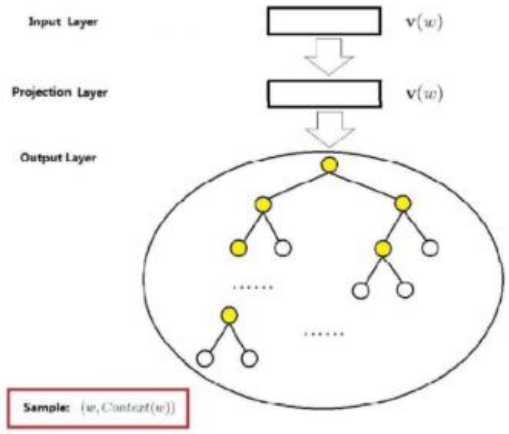


图 12 Skip-gram 模型的网络结构示意图

输入层：只含当前样本的中心词 w 的词向量$v(w)\epsilon R^m$

投影层： 恒等投影，应该去除（没用）

输出层：输出层对应一颗二叉树，它是以预料库中出现过的词当作叶子结点，以各词在语料库中出现的次数当作权值构建出来的 Huffman 树。该树中叶子结点有|D|个，D 为字典大小；非叶子结点有|D|-1 个。

对于层次 SoftMax 的 skip-gram 模型，已知的是当前词 w，需要对其上下文 Context(w)中的词进行预测，故，其目标函数为：

$$\begin{aligned} L &= \sum_{w\epsilon C} \log p(Context(w)|w) = \sum_{w\epsilon C} \log \prod_{u\epsilon Context(w)} p(u|w) \\ &= \sum_{w\epsilon C} \log \prod_{u\epsilon Context(w)} \prod_{j=2}^{l^u} \{[\sigma(v(w)^T\theta_{j-1}^u)]^{(1-d_j^u)} \cdot [1-\sigma(v(w)^T\theta_{j-1}^u)]^{d_j^u} \\ &= \sum_{w\epsilon C} \sum_{u\epsilon Context(w)} \sum_{j=2}^{l^u} \{(1-d_j^u)\cdot log[\sigma(v(w)^T\theta_{j-1}^u)] + d_j^u \cdot log[1-\sigma(v(w)^T\theta_{j-1}^u)]\} \end{aligned}$$

负采样

Negative sampling（NEG）可以视为对层次 SoftMax 的一种代替，它不再使用（复杂的）Huffman 树，而是利用（相对简单的）随机负采样，其目的也是用来提高训练速度并改善所得词向量的质量。

Training a neural network means taking a training example and adjusting all of the neuron weights slightly so that it predicts that training sample more accurately. In

other words, each training sample will tweak all of the weights in the neural network.

As we discussed above, the size of our word vocabulary means that our neural network has a tremendous number of weights, all of which would be updated slightly by every one of our billions of training samples!

Negative sampling addresses this by having each training sample only modify a small percentage of the weights, rather than all of them. Here's how it works.

When training the network on the word pair ("fox", "quick"), recall that the "label" or "correct output" of the network is a one-hot vector. That is, for the output neuron corresponding to "quick" to output a 1, and for all of the other thousands of output neurons to output a 0.

With negative sampling, we are instead going to randomly select just a small number of "negative" words (let's say 5) to update the weights for. (In this context, a "negative" word is one for which we want the network to output a 0 for). We will also still update the weights for our "positive" word (which is the word "quick" in our current example).

The paper says that selecting 5-20 words works well for smaller datasets, and you can get away with only 2-5 words for large datasets.
Recall that the output layer of our model has a weight matrix that's 300 x 10,000. So we will just be updating the weights for our positive word ("quick"), plus the weights for 5 other words that we want to output 0. That's a total of 6 output neurons, and 1,800 weight values total. That's only 0.06% of the 3M weights in the output layer!

In the hidden layer, only the weights for the input word are updated (this is true whether you're using Negative Sampling or not).

Word2vec 采用带权采样法：词典 D 中的词在语料 C 中出现的次数有高有底，对于那些高频词，被选为负样本的概率就应该比较大，而低频词出现的概率应该比较小。Word2vec 源码中使用下面的公式设置每个词的权重：

$$\text{len(w)} = \frac{[counter(w)]^{0.75}}{\sum_{w \epsilon D}[counter(w)]^{0.75}}$$

在 CBOW 模型中，已知词 w 的上下文 Context(w)，需要预测 w，因此，对于给定的上下文 Context(w)，词 w 就是一个正样本，其他的词就是负样本。负样本太多了，因此我们一般通过上述方法选取适量的负样本 $\mathrm{NEG(w)}$

假定现在已经选好了一个关于 Context(w)的负样本子集 $\mathrm{NEG(w)}$

对于一个给定的正样本 $(Context(w), w)$, 我们希望最大化

$$p(u|Context(w)) = \begin{cases} \sigma(\mathbf{x}_w^\top \theta^u), \\ 1-\sigma(\mathbf{x}_w^\top \theta^u) \end{cases}$$

$$g(w) = \prod_{u \in \{w\} \cup NEG(w)} p(u|Context(w))$$

正例　　1-负例=正例

$$g(w) = \sigma(\mathbf{x}_w^\top \theta^w) \prod_{u \in NEG(w)} [1-\sigma(\mathbf{x}_w^\top \theta^u)]$$

← 希望最终的结果朝着正确的方向发展

$\sigma(\mathbf{x}_w^\top \theta^w)$ 表示当上下文为 $Context(w)$ 时, 预测中心词为 w 的概率

$\sigma(\mathbf{x}_w^\top \theta^u)$, $u \in NEG(w)$ 则表示当上下文为 $Context(w)$ 时, 预测中心词为 u 的概率

负样本空间

对于一个给定的语料库 $\mathcal{C}$

$$G = \prod_{w \in \mathcal{C}} g(w)$$

$$\begin{aligned} \mathcal{L} &= \log G = \log \prod_{w \in \mathcal{C}} g(w) = \sum_{w \in \mathcal{C}} \log g(w) \\ &= \sum_{w \in \mathcal{C}} \log \prod_{u \in \{w\} \cup NEG(w)} \left\{ [\sigma(\mathbf{x}_w^\top \theta^u)]^{L^w(u)} \cdot [1-\sigma(\mathbf{x}_w^\top \theta^u)]^{1-L^w(u)} \right\} \\ &= \sum_{w \in \mathcal{C}} \sum_{u \in \{w\} \cup NEG(w)} \left\{ L^w(u) \cdot \log [\sigma(\mathbf{x}_w^\top \theta^u)] + [1-L^w(u)] \cdot \log [1-\sigma(\mathbf{x}_w^\top \theta^u)] \right\} \end{aligned}$$

$$\begin{aligned} \frac{\partial \mathcal{L}(w,u)}{\partial \theta^u} &= \frac{\partial}{\partial \theta^u} \left\{ L^w(u) \cdot \log [\sigma(\mathbf{x}_w^\top \theta^u)] + [1-L^w(u)] \cdot \log [1-\sigma(\mathbf{x}_w^\top \theta^u)] \right\} \\ &= L^w(u)[1-\sigma(\mathbf{x}_w^\top \theta^u)]\mathbf{x}_w - [1-L^w(u)]\sigma(\mathbf{x}_w^\top \theta^u)\mathbf{x}_w \\ &= \left\{ L^w(u)[1-\sigma(\mathbf{x}_w^\top \theta^u)] - [1-L^w(u)]\sigma(\mathbf{x}_w^\top \theta^u) \right\} \mathbf{x}_w \\ &= \left[L^w(u) - \sigma(\mathbf{x}_w^\top \theta^u) \right] \mathbf{x}_w \end{aligned}$$

θ^u 的更新公式可写为 $\theta^u := \theta^u + \eta \left[L^w(u) - \sigma(\mathbf{x}_w^\top \theta^u)\right] \mathbf{x}_w$

$$\frac{\partial \mathcal{L}(w,u)}{\partial \mathbf{x}_w} = \left[L^w(u) - \sigma(\mathbf{x}_w^\top \theta^u)\right] \theta^u$$

$$\mathbf{v}(\tilde{w}) := \mathbf{v}(\tilde{w}) + \eta \sum_{u \in \{w\} \cup NEG(w)} \frac{\partial \mathcal{L}(w,u)}{\partial \mathbf{x}_w}, \quad \tilde{w} \in Context(w)$$

最终词向量结果

对于 Negative sampling 的 skip-gram 模型,其目标函数：

$$\begin{aligned}
\mathcal{L} &= \log G = \log \prod_{w \in \mathcal{C}} g(w) = \sum_{w \in \mathcal{C}} \log g(w) \\
&= \sum_{w \in \mathcal{C}} \log \prod_{\tilde{w} \subset Context(w)} \prod_{u \subset \{w\} \cup NEG^{\tilde{w}}(w)} \left\{ \left[\sigma(\mathbf{v}(\tilde{w})^\top \theta^u)\right]^{L^w(u)} \cdot \left[1 - \sigma(\mathbf{v}(\tilde{w})^\top \theta^u)\right]^{1 - L^w(u)} \right\} \\
&= \sum_{w \in \mathcal{C}} \sum_{\tilde{w} \in Context(w)} \sum_{u \in \{w\} \cup NEG^{\tilde{w}}(w)} \\
&\quad \left\{ L^w(u) \cdot \log \left[\sigma(\mathbf{v}(\tilde{w})^\top \theta^u)\right] + \left[1 - L^w(u)\right] \cdot \log \left[1 - \sigma(\mathbf{v}(\tilde{w})^\top \theta^u)\right] \right\}.
\end{aligned}$$

Doc2Vec

heavily based on word2vec

Word2vec 训练词向量的细节，讲解了一个词是如何通过 word2vec 模型训练出唯一的向量来表示的。那接着可能就会想到，有没有什么办法能够将一个句子甚至一篇短文也用一个向量来表示呢？答案是肯定有的，构建一个句子向量有很多种方法：

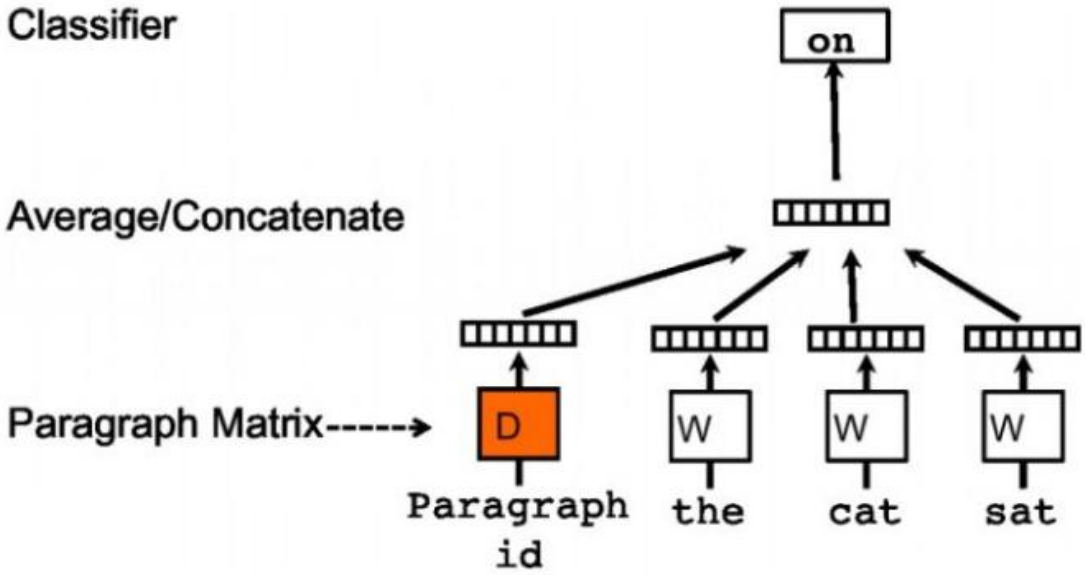
1. 词频
2. TF-IDF
3. 词向量平均
4. Doc2Vec

许多机器学习算法需要的输入是一个固定长度的向量，当涉及到短文时，最常用的固定长度的向量方法是词袋模型（bag-of-words）。尽管它很流行，但是词袋模型存在两个主要的缺点：一个是词袋模型忽略词序，如果两个不同的句子由相同的词但是顺序不同组成，词袋模型会将这两句话定义为同一个表达；另一个是词袋模型忽略了句法，这样训练出来的模型会造成类似‘powerful‘，‘strong’和‘Paris’的距离是相同的，而其实‘powerful’应该相对于’ Paris ‘距离’ strong ‘更近才对。

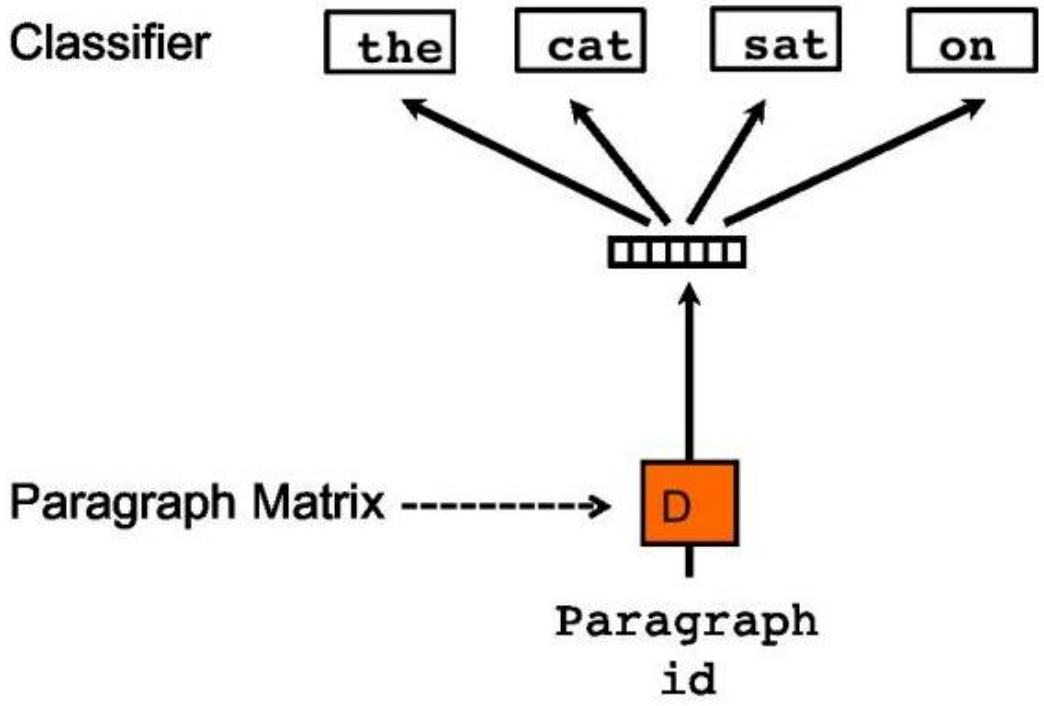
Doc2vec 又叫 Paragraph Vector 是 Tomas Mikolov 基于 word2vec 模型提出的，其

具有一些优点，比如不用固定句子长度，接受不同长度的句子做训练样本，Doc2vec 是一个无监督学习算法，该算法用于预测一个向量来表示不同的文档，该模型的结构潜在的克服了词袋模型的缺点。

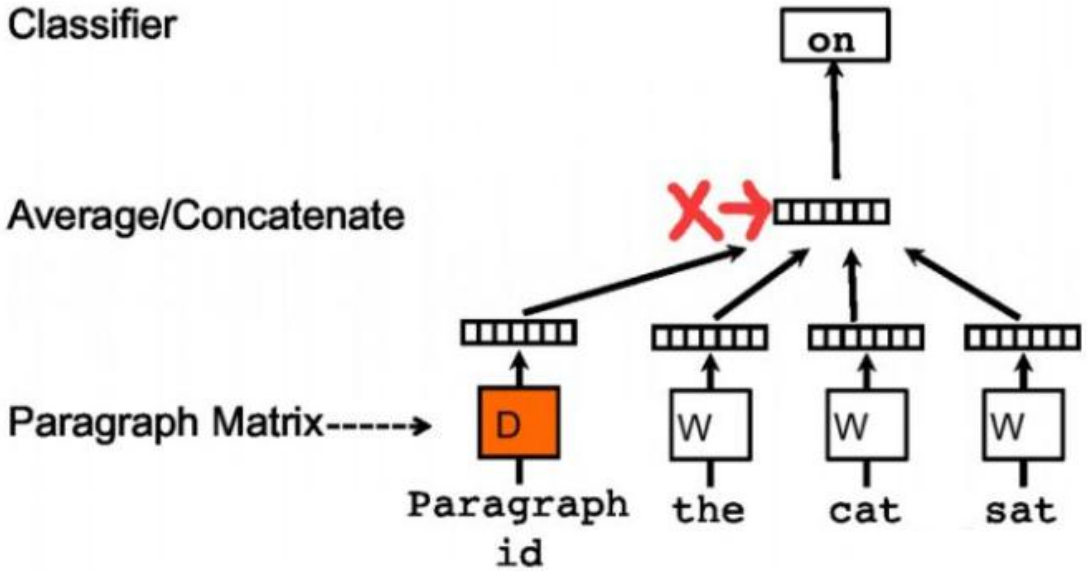
Doc2vec 模型是受到了 word2vec 模型的启发，word2vec 里预测词向量时，预测出来的词是含有词义的，比如上文提到的词向量‘ powerful’会相对于‘ Paris’离‘ strong ‘距离更近，在 Doc2vec 中也构建了相同的结构。所以 Doc2vec 克服了词袋模型中没有语义的缺点。假设现在存在训练样本，每个句子是训练样本。和 word2vec 一样，Doc2vec 也有两种训练方式，一种是 PV-DM(Distributed Memory Model of paragraph vectors ）类似于 word2vec 中的 CBOW 模型，如图一：



另一种是 PV-DBOW(Distributed Bag of Words of paragraph vector 类似于 Word2vec 中的 skip-gram 模型，如图二：



在 Doc2vec 中，每一句话用唯一的向量来表示，用矩阵 D 的某一列来代表。每一个词也用唯一的向量来表示，用矩阵 W 的某一列来表示。以 PV-DM 模型为例，如图三：



每次从一句话中滑动采样固定长度的词，取其中一个词作预测词，其他的作输入词。输入词对应的词向量 word vector 和本句话对应的句子向量 Paragraph vector 作为输入层的输入，将本句话的向量和本次采样的词向量相加求平均或者累加构成一个新的向量 X，进而使用这个向量 X 预测此次窗口内的预测词。

Doc2vec 相对于 word2vec 不同之处在于，在输入层，增添了一个新的句子向量 Paragraph vector，Paragraph vector 可以被看作是另一个词向量，它扮演了一个记忆，词袋模型中，因为每次训练只会截取句子中一小部分词训练，而忽略了除了本次训练词以外该句子中的其他词，这样仅仅训练出来每个词的向量表达，句子只是每个词的向量累加在一

起表达的。正如上文所说的词袋模型的缺点，忽略了文本的词序问题。而 Doc2vec 中的 Paragraph vector 则弥补了这方面的不足，它每次训练也是滑动截取句子中一小部分词来训练，Paragraph Vector 在同一个句子的若干次训练中是共享的，所以同一句话会有多次训练，每次训练中输入都包含 Paragraph vector。它可以被看作是句子的主旨，有了它，该句子的主旨每次都会被放入作为输入的一部分来训练。这样每次训练过程中，不光是训练了词，得到了词向量。同时随着一句话每次滑动取若干词训练的过程中，作为每次训练的输入层一部分的共享 Paragraph vector，该向量表达的主旨会越来越准确。Doc2vec 中 PV-DM 模型具体的训练过程和 word2vec 中的 CBOW 模型训练方式相同，在之前有详细介绍，这里就不在重复。

训练完了以后，就会得到训练样本中所有的词向量和每句话对应的句子向量，那么 Doc2vec 是怎么预测新的句子 Paragraph vector 呢？其实在预测新的句子的时候，还是会将该 Paragraph vector 随机初始化，放入模型中再重新根据随机梯度下降不断迭代求得最终稳定下来的句子向量。不过在预测过程中，模型里的词向量还有投影层到输出层的 softmax weights 参数是不会变的，这样在不断迭代中只会更新 Paragraph vector，其他参数均已固定，只需很少的时间就能计算出带预测的 Paragraph vector。

在 python 中使用 gensim 包调用 Doc2vec 方便快捷。

小结

Doc2vec 是基于 Word2vec 基础上构建的，相比于 Word2vec，Doc2vec 不仅能训练处词向量还能训练处句子向量并预测新的句子向量。Doc2vec 模型结构相对于 Word2vec，不同点在于在输入层上多增加了一个 Paragraph vector 句子向量，该向量在同一句下的不同的训练中是权值共享的，这样训练出来的 Paragraph vector 就会逐渐在每句子中的几次训练中不断稳定下来，形成该句子的主旨。这样就训练出来了我们需要的句子向量。

在预测新的句子向量时，是需要重新训练的，此时该模型的词向量和投影层到输出层的 soft weights 参数固定，只剩下 Paragraph vector 用梯度下降法求得，所以预测新句子时虽然也要放入模型中不断迭代求出，相比于训练时，速度会快得多。