Walk Through Word2Vec Source Code

在阅读这篇文章之前，请确保对神经网络和梯度下降有一定的了解，否则对于第四部分的推导可能会很吃力，具体可参考我之前写的两篇文章Gradient Descent Example Code和Simple Neural Network.

1 初始化工作

1.1 Subsampling

假如有一个句子，单词a，b，c，d，e，f分别出现1，2，3，4，6，100次，那么采用subsampling方式，f极有可能被去掉，因为subsampling会去除词频过高的词，而往往词频较高的单词都是一些停用词，去掉合情合理.

1.2 构建词库

通过subsampling之后，现有单词a，b，c，d，e分别出现1，2，3，4，6次，通过词频，按照升序排列，得到如下词库:

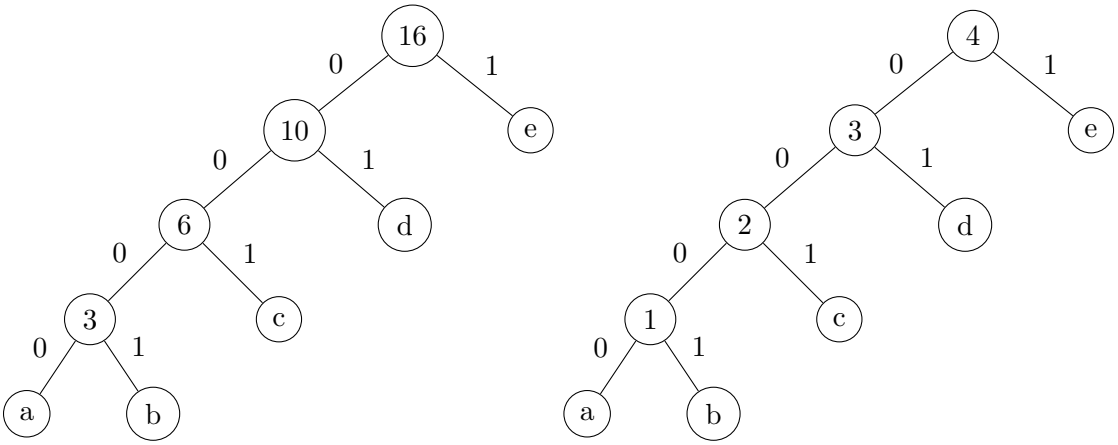
WORD	a	b	c	d	e
COUNT	1	2	3	4	6

再依照词频对词库进行排序，并且对每一个词进行编号，得到新词库:

ID	1	2	3	4	5
WORD	a	b	c	d	e

1.3 构建Huffman Tree

得到排序后的词库之后，我们会通过词频构建Huffman Tree:



左边的Huffman树是最初构建的，还需要处理. 我们把左边树中，非叶子节点按照值的升序进行编号，使得每一个非叶子节点都有ID，就变成了右边我们想要的Huffman树.

这里提醒一点，词库中每个单词，对应到Huffman Tree的叶子节点，都有ID；Huffman Tree上，每个非叶子节点也有ID，是两套不同的ID编号，都从1开始，前者最大ID为单词数，后者最大ID为非叶子节点数，其实我们通过观察Huffman Tree，可以发现前者最大ID比后者大1，即叶子节点比非叶子节点多1.

2 CBoW(Continuous Bag of Words)模型

这一章我们讲一讲CBow模型，简单地说，就是给你一个句子，把其中一个词去掉，你通过训练模型，预测词库里哪个词最有可能.

2.1 Input Layer to Hidden Layer

以单词d作为预测单词，d在Huffman中，父节点自上而下id为4-3，对应的编码0-1，与d相邻的单词为b，c. 单词d在词库中的索引号应该为4，用one-hot表示则为[0 0 0 1 0]，而相邻单词b，c分别为[0 1 0 0 0]，[0 0 1 0 0]，在CBoW中，会将b，c的one-hot向量相加，即[0 1 1 0 0]，这里我们通过d周围的单词来训练模型，使得模型输出d的Huffman编码，即预测 d 最为可能是b，c中缺失的词.

这里首先计算从输入层到隐含层neu1，定义输入层与隐含层之间的连接权矩阵syn0，其实这个syn0就是我们最终要求的词向量矩阵，每一行为一个词的词向量，行号对应词库中词的ID号，这里词向量维度为6:

$$syn0_{5*6} = \begin{bmatrix} syn0_{11} & \dots & syn0_{16} \\ \dots & \dots & \dots \\ syn0_{51} & \dots & syn0_{56} \end{bmatrix} \tag{2.1.1}$$

这里做矩阵相乘，输入层为向量[0 1 1 0 0]，乘以连接权，得到隐含层，Word2Vec中隐含层不做激活操作:

$$neu0_{1*6} = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} syn0_{11} & \dots & syn0_{16} \\ \dots & \dots & \dots \\ syn0_{51} & \dots & syn0_{56} \end{bmatrix} = \begin{bmatrix} neu1_{11} & \dots & neu1_{16} \end{bmatrix} \tag{2.1.2}$$

2.2 Hidden Layer to Output Layer

再定义隐含层与输出层之间的连接权矩阵syn1，这里需要注意的是syn1的列数，即syn1的行空间维度跟我们构建的Huffman Tree的非叶子节点个数必须一致，通过后面的讲解，读者就会理解，其实可以把syn1的每一列看成是父节点(即Huffman Tree中非叶子节点)的词向量，这里稍微有点抽象，因为Huffman Tree中叶子节点是实实在在的单词，而非叶子节点没有具体的单词，存的只是子节点词频之和:

$$syn1_{6*4} = \begin{bmatrix} syn1_{11} & \dots & syn1_{14} \\ \dots & \dots & \dots \\ syn1_{61} & \dots & syn1_{64} \end{bmatrix} \tag{2.2.1}$$

这里做矩阵相乘，输入为隐含层neu1，与连接矩阵syn1相乘:

$$y_{1*4} = \begin{bmatrix} neu1_{11} & \dots & neu1_{16} \end{bmatrix} \cdot \begin{bmatrix} syn1_{11} & \dots & syn1_{14} \\ \dots & \dots & \dots \\ syn1_{61} & \dots & syn1_{64} \end{bmatrix} = \begin{bmatrix} y_1 & y_2 & y_3 & y_4 \end{bmatrix} \tag{2.2.2}$$

以上是Softmax的计算方式，即将每一个词的可能性计算出来，但不是Word2Vec中提到的Hierarchical Softmax. Word2Vec会根据缺失词(我们训练的时候，肯定知道哪个词是缺失的)的Huffman中父节点ID，从syn1中选出相对应的列向量，与隐含层相乘，以下是完整过程，有一点需要说明的是，如果Huffman中的编码为0-1，Word2Vec要求预测的结果为1-0，刚好跟现实编码相反，需要读者稍微做一点转换:

单词d第一个父节点ID为4，则从syn1中取出第4列向量:

$$y_4 = \begin{bmatrix} neu1_{11} & \dots & neu1_{16} \end{bmatrix} \cdot \begin{bmatrix} syn1_{14} \\ \dots\dots \\ syn1_{64} \end{bmatrix} \tag{2.2.3}$$

这里的y_4为Huffman路径预测，父节点ID为4，对应的Huffman编码为0，即需要y_4为1，我们可以通过不断训练syn0与syn1参数实现.

第二个父节点ID为3，则从syn1中取出第3列向量

$$y_3 = \begin{bmatrix} neu1_{11} & \dots & neu1_{16} \end{bmatrix} \cdot \begin{bmatrix} syn1_{13} \\ \dots\dots \\ syn1_{63} \end{bmatrix} \tag{2.2.4}$$

这里的y_3同样为Huffman路径预测，父节点ID为3，对应的Huffman编码为1，即需要y_3为0.

通过上面的计算，我们可以发现，最终计算得出的Huffman路径为0-1，比传统的Softmax少了2次计算。同时，我们发现我们是通过Huffman Tree父节点ID从syn1中选择相应的列向量进行计算，这也说明了为什么syn1行空间维度与Huffman Tree非叶子节点个数一致.

2.3 Negative Sampling

上面我们通过正确的样本训练模型，相当于我们只告诉他怎么计算是对的，如果采用Negative Sampling的思想，我们还应该告诉模型怎么计算是错的，尽量使得y_1与y_2的输出等于0，这样一定程度上避免了太多不确定性. 因此会随机的计算几个y_x，使得y_x=0，这个步骤跟上面计算y_3，y_4一样，当然，隐含层与输出层之间的权重就变成了syn1neg，这里syn1与syn1neg结构完全一致. 这里留一个疑问，为什么要再加一个权重矩阵syn1neg，为什么不使用原先的syn1？这个问题后面章节讨论，需要读者对Word2Vec有一定的掌握之后，才能更好的理解.

3 Skip-Gram模型

其实Skip-Gram跟CBoW非常的相似，但在计算的过程中我们稍微做一点思维上的转换. 从上面CBoW模型的计算过程来看，我们通过缺失词的周边几个词作为输入层，而最终预测的标签为缺失词的Huffman路径. Skip-Gram模型训练过程中，缺失词即需要预测的词是已知词的周边几个词，按照CBoW的逻辑，这里我们不妨做个假设，是不是用已知词作为输入，每一个缺失词的Huffman路径作为最终需要预测的结果?

其实恰恰相反，Skip-Gram采用的方法是使用周边的几个词作为输入，训练相关的参数，使得预测的结果刚好为已知词的Huffman路径. 只不过，这里有一点需要明确，在Skip-Gram中，输入不同于CBoW，不是简单的将周围词的one-hot向量叠加，而是采用逐个训练的方式训练模型. 假如我们已知词d，需要预测周围词b，c. 那面输入层有两个向量，即[0 1 0 0 0]和[0 0 1 0 0]，重复公式2.1.1，2.1.2，2.2.3，2.2.4的操作，Negative Sampling方法同CBoW一致.

这里需要明确的一点，CBoW和Skip-Gram训练出的词向量是不一样的. 另外，训练CBoW速度要快很多，因为CBoW是临近的词向量相加训练，而Skip-Gram是逐词训练. 当然，Skip-Gram虽然慢，但通过大量的训练，保留了更多的语义.

4 关于Word2Vec中模型的训练–进阶

其实大家也可以发现，Word2Vec采用的是前馈神经网络，回顾上一篇文章Simple Neural Network，我们介绍了BP(Back Propagation)算法，很多神经网络都使用BP算法来训练参数，最关键的就是理解链式求导法则，通过求导我们可以算出梯度，有了梯度，我们就可以使用梯度下降的方式来训练模型，常用的梯度下降方法可以看之前一篇Gradient Descent Example Code. 而Word2Vec采用的就是BP算法，只不过，跟先前我介绍的BP算法有一些差异.

在Word2Vec模型训练中，其实用到了两种梯度下降法，即SGD(Stochastic Gradient Descent)和MGD(Mini-Batch Gradient Descent)，在训练syn0，syn1时，分别是用了MGD和SGD. 下面我们讲讲Word2Vec参数的训练.

4.1 Train syn1

之所以把syn1的训练提到syn0之前，是因为BP算法是向后传递误差，因此第一个更新的参数矩阵是syn1. 既然我们前面降到了syn1的训练使用到了SGD，我们是如何更新syn1的呢? 观察公式2.2.3，2.2.4，每一个预测都可能跟真实值有偏差，那么每计算一个y_x，Word2Vec就用相应的误差来训练syn1.

具体的更新代码为:

```
g = (1 - vocab[word].code[d] - f) * alpha; // vocab[word].code[d] is the dth parent node ID of
    target word
l2 = vocab[word].point[d] * layer1_size;
for (c = 0; c < layer1_size; c++) syn1[c + l2] += g * neu1[c];
```

再次强调下，Word2Vec要求模型预测的路径刚好跟Huffman编码相反，即编码为0-1，则预测为1-0.

第一行其实只是一个中间值，不做讨论.

第二行代码中$l2$为syn1中，第$parent_node = vocab[word].point[d]$列的起始地址，即对应单词word的第d个父节点词向量syn1[*][parent_node](这是一个列向量). 这里syn1中的参数是一列一列存的，目的是为了方便计算，因为我们在公式2.2.3和2.2.4，都读取了相关的列.

第三行$g * neu1 = (1 - vocab[word].code[d] - f) * alpha * neu1$，其中$(1 - vocab[word].code[d] - f) * neu1$可认

为是负梯度，但不是标准的按照梯度链式求导法则得出的结果，因为损失函数关于syn1求导，标准的负梯度为$(1 - vocab[word].code[d] - f) * f * (1 - f) * neu1$，这里$f * (1 - f) \leq 0.25$，因此如上代码给出的负梯度值会过大，但是观察代码，我们发现步长alpha是一个比较小的值，且随着迭代次数，本身会逐渐减小，一定程度上对梯度做了一些修正.

继续观察第三行代码，我们可以发现，这行代码用于更新syn1的vocab[word].point[d]列. 我们之前在讲BP的时候都是更新syn1完整的矩阵，为什么这里就变成了更新一列，其实也好理解，因为公式2.2.3和2.2.4中，只有相关列参与了计算，我们要做的只是更新相关的列.

4.2 Train syn0

我们前面讲到了syn0的训练使用到了MGD，那么一个Mini Batch是多少个误差累加呢？这里先不讨论Negative Sampling，通过代码我们发现，Word2Vec中将公式2.2.3，2.2.4的误差累加起来，用于训练syn1，即假如某个单词Huffman路径长度为n，则一个Mini Batch中累加了n个预测的误差.

具体更新代码:

```
1  for (d = 0; d < vocab[word].codelen; d++)
2  {
3    g = (1 - vocab[word].code[d] - f) * alpha; // vocab[word].code[d] is the dth parent node ID of
       missing word
4    l2 = vocab[word].point[d] * layer1_size;
5    for (c = 0; c < layer1_size; c++) neu1e[c] += g * syn1[c + l2];
6  }
7
8  for (a = b; a < window * 2 + 1 - b; a++)
9  {
10   if (a == window) continue;// the missing word itself
11
12   last_word = sen[c]; // neighbor word
13   l2 = last_word * layer1_size;
14   for (c = 0; c < layer1_size; c++) syn0[c + l2] += neu1e[c];
15 }
```

先看第14行代码，变量$neu1e$其实是前面循环累加的误差. 词的Huffman路径长度$vocab[word].codelen$，决定了累加的次数. 结合先前的例子，输入层[0 1 1 0 0]，计算相应的损失函数，然后使用BP算法更新相对应的词向量$syn0$. 令$\gamma_i = (1 - y_4) * y_4 * (1 - y_4) * syn1_{i4} + (1 - y_3) * y_3 * (1 - y_3) * syn1_{i3}$，根据BP公式:

$$syn0' = syn0 + alpha * \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \end{bmatrix} \cdot (\begin{bmatrix} \gamma_1 & \gamma_1 & \gamma_1 & \gamma_1 \end{bmatrix} * neu1 * (1 - neu1))) \tag{4.2.1}$$

$$= syn0 + alpha * \begin{bmatrix} 0 & 0 & 0 & 0 \\ t_1 & t_2 & t_3 & t_4 \\ t_1 & t_2 & t_3 & t_4 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \ where \ t_i = \gamma_i * neu1_i \tag{4.2.2}$$

我们发现，其实我们只更新了$syn0[2][*]$和$syn0[3][*]$，即syn0中第 2 ，3个词向量. 我们可以改写成:

$$syn0'_{2*} = syn0_{2*} + alpha * \begin{bmatrix} \gamma_1 & \gamma_2 & \gamma_3 & \gamma_4 \end{bmatrix} * neu1 * (1 - neu1) \tag{4.2.3}$$

$$syn0'_{3*} = syn0_{3*} + alpha * \begin{bmatrix} \gamma_1 & \gamma_2 & \gamma_3 & \gamma_4 \end{bmatrix} * neu1 * (1 - neu1) \tag{4.2.4}$$

以上是严格按照BP算法的更新方式，我们再看看Word2Vec代码是如何更新，以更新$syn0_{2*}$为例，转换成公式，可写成如下:

$$syn0'_{2*} = syn0_{2*} + alpha * \begin{bmatrix} \gamma_1 & \gamma_2 & \gamma_3 & \gamma_4 \end{bmatrix}) \tag{4.2.5}$$

不难发现，代码做了简化，将γ_i中的$y_4 * (1 - y_4)$，$y_3 * (1 - y_3)$去掉，另外$neu1 * (1 - neu1)$也舍去，之前我们在训练syn1时讨论过，通过变换步长，去掉$y_4 * (1 - y_4)$和$y_3 * (1 - y_3)$是允许的，那么$neu1 * (1 - neu1)$呢? 其实这是有问题的，$neu1 * (1 - neu1)$是隐含层梯度的方向，作为整体梯度的一部分，舍去会影响整体梯度的方向(这里有疑问可以回顾下链式求导的知识). 如果消去，可理解为隐含层梯度方向变成为[1 1 1 1]，那么，这根本无法寻找最优解.

下面我通过简单的例子来证明我的判断，下面的代码是之前Simple Neural Network中的一个BP算法样例，只是对迭代次数做了修改:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

"""simple-neural-network.py: Train a model that outputs 1 when the first
and second unit of a sample is the same value, otherwise outputs 0. """

__author__    = "Cai-Jun Sun"
__copyright__ = "Copyright 2016, IoVision"
__license__   = "GPL"
__version__   = "1.0.0"
__maintainer__= "Cai-Jun Sun"
__email__     = "sun(dot)caijun(at)bupt(dot)edu(dot)cn"
__status__    = "Production"

import numpy as np

def sigmod(x):
    return 1/(1+np.exp(-x))

def test(x):
    l1 = sigmod(np.dot(x,syn0))
    return sigmod(np.dot(l1,syn1))

step_size = 1 # step size for gradient descent
iteration_count = 1000

X = np.array([[0,0,0], [0,0,1], [0,1,0], [0,1,1], [1,0,0], [1,0,1], [1,1,0]])
Y = np.array([[1,1,0,0,0,0,1]]).T

np.random.seed(1)

# random initial weights, ranging from -0.5 ~ 0.5
```

```
33 syn0 = 2 * np.random.random((3,15)) - 1
34 syn1 = 2 * np.random.random((15,1)) - 1
35
36 for j in xrange(iteration_count):
37     Samples = X # input layer
38     Hidden = sigmod(np.dot(Samples,syn0)) # hidden layer
39     Output = sigmod(np.dot(Hidden,syn1)) # output layer
40
41     syn1 += (-1) * step_size * Hidden.T.dot((Output - Y) * Output * (1 - Output))
42     syn0 += (-1) * step_size * Samples.T.dot(((Output - Y) * Output * (1 - Output))\
43                                             .dot(syn1.T) * Hidden * (1-Hidden))
44
45 print test([1,1,1]) # output 0.48776692
46 print test([2,2,2]) # output 0.80574699
47 print test([3,3,3]) # output 0.86762845
```

按照word2vec的写法，对42行代码做了相应的修改:

```
1 syn0 += (-1) * step_size * Samples.T.dot(((Output - Y) * Output * (1 - Output))\
2                                         .dot(syn1.T))
```

通过运行，我们发现根本无法训练模型，那么Word2Vec的写法是否真的存在问题? 其实没有，我们忘了最关键的一点，即隐藏函数没有做激活，输入层到隐含层根本就只是做了一个线性转换，因此，隐含层求导的梯度方向就是[1 1 1 1]，所以Word2Vec的代码是完全正确的.

4.3 Train with Negative Sampling

上面我们在更新syn1和syn0时，没有考虑Negative Sampling，那么现在讲讲加入Negative Sampling后的代码:

```
1  for (d = 0; d < negative + 1; d++)
2  {
3    if (d == 0)
4    {
5      target = word;
6      label = 1; // the value can be whatever you like
7    }else
8    {
9      next_random = next_random * multiplier + 11;
10     target = table[(next_random >> 16) % table_size]; // table_size is 1e8
11
12     if (target == 0) target = next_random % (vocab_size - 1) + 1; // target 0 is '<s>'
13     if (target == word) continue;
14
15     label = 0; // the value can be whatever you like
16   }
17
18   l2 = target * layer1_size;
19   f = 0;
20   for (c = 0; c < layer1_size; c++) f += neu1[c] * syn1neg[c + l2];
21
22   // the prediction label is expected to be 0 as we define above
23   if (f > MAX_EXP) g = (label - 1) * alpha;
24   else if (f < -MAX_EXP) g = (label - 0) * alpha;
25   else g = (label - expTable[(int)((f + MAX_EXP) * (EXP_TABLE_SIZE / MAX_EXP / 2))]) * alpha;
26
```

```
27   for (c = 0; c < layer1_size; c++) neu1e[c] += g * syn1neg[c + l2];
28   for (c = 0; c < layer1_size; c++) syn1neg[c + l2] += g * neu1[c];
29 }
```

其实，上面的$syn1neg$跟$syn1$是一毛一样的，只不过后者用于训练正确样本，而前者用于训练错误样本，syn1把词向量映射成正确的Huffman编码，而syn1neg统统映射到错误编码，即0-1-1-1-1，代码里的label与Huffman的编码刚好相反，label为1-0-0-0-0，这个label序列的长度跟用户定义的negative大小有关. 后续的更新$syn1neg$和$syn0$代码是一样的.

接着2.3中提到的问题，这里我们谈一谈为什么需要一个额外的syn1neg，而不是在原来syn1基础上继续训练. 其实我们先前提到了，syn0的每一行是词向量，syn1的每一列是父节点的词向量. 以Skip-Gram为例，假如已知词Huffman Tree父节点ID为x，y，对应的父节点词向量syn1[*][x](列向量)，syn1[*][y](列向量)，Huffman编码为0-1，周围两个词的词向量syn0[i][*](行向量)，syn0[j][*](行向量)，我们在训练的时候，要求满足:

$$sigmod(syn0[i][*] \cdot syn1[*][x]) \approx 1$$
$$sigmod(syn0[j][*] \cdot syn1[*][x]) \approx 1$$
$$sigmod(syn0[i][*] \cdot syn1[*][y]) \approx 0$$
$$sigmod(syn0[j][*] \cdot syn1[*][y]) \approx 0$$

我们再看看Negative Sampling，如果只是在syn1上更新，它会把$sigmod(syn0[u][*] * syn1[*][v])$的大部分结果变成0，少部分变成1，那么我们上边的条件很难满足，即这是一个来回震荡的过程，上面的好不容易把参数训练好，Negative Sampling又把他给掰弯了. 因此，采用一个新的连接权矩阵，能做到相互不干扰.

上面我们以Skip-Gram为例，说明还得有一个syn1neg，那么对于CBoW模型训练呢? 其实也需要，假如缺失词Huffman Tree父节点ID为x，y，对应的父节点词向量syn1[*][x](列向量)，syn1[*][y](列向量)，Huffman编码为0-1，周围两个词的词向量syn0[i][*](行向量)，syn0[j][*](行向量)，CBoW计算时，要求满足:

$$sigmod((syn0[i][*] + syn0[j][*]) \cdot syn1[*][x]) \approx 1$$
$$sigmod((syn0[i][*] + syn0[j][*]) \cdot syn1[*][y]) \approx 0$$

一旦Negative Sampling把$sigmod(syn0[u][*]*syn1[*][v])$的结果训练成0，$sigmod((syn0[i][*]+syn0[j][*])\cdot syn1[*][x])$的结果也会变成0. 因此，两种模型训练时，都需要额外添加一个syn1neg权重矩阵，不至于相互干扰.

5 Future Work

关于Word2Vec的注解，可看我的博客word2vec.c，里面有比较详细的解释，当然，对原作者的代码做了一些格式做了些调整，并且对一些函数和变量做了重命名，以方便读者理解. 后面我们将用最简单的代码，实现一个toy Word2Vec.