Word2Vec理论与实践

Word2vec的作用是将所有词语投影到K维的向量空间，每个词语都可以用一个K维向量表示。它同时是Embedding中非常基本的模型，训练出来的词向量不仅能保持语义与语法上的相关性，并且可以实现类似代数运算的能力。由于它简洁，高效的特点，引起了人们的广泛关注，并应用在很多NLP任务中，用于训练相应的词向量。

# One-Hot encoding

One-hot representation把每个词表示为一个向量。这个向量的维度是词表大小，只有一个维度的值为 1，这个维度就代表了当前的词。

比如：词典是{ 'I', 'have', 'an', 'apple' }，那么词典中各个单词的one-hot表示如下：

|  |
| --- |
| V(I) = [ 1, 0, 0, 0 ]  V(have) = [ 0, 1, 0, 0 ]  V(an) = [ 0, 0, 1, 0 ]  V(apple) = [ 0, 0, 0, 1 ] |

通过计数word在文档中出现的次数进行编码，还有另一种表示，被称为count vectorizing。假设词典共有V个词，那么每一个单词都有一个V维度的向量来表示，向量中只有一个位置为非0（可以是1，也可以是出现次数），表示该词的编号，其余为0。

采用scikit-learn实现one-hot的代码见github：

|  |
| --- |
| from sklearn.feature\_extraction.text import CountVectorizer  vectorizer = CountVectorizer()  corpus = [  'Text of first document.',  'Text of the second document make longer.',  'Number three.',  'This is number four.',  ]  # store CountVectorizer sparse matrix in X  # The column of matrix is words, rows are documents  X = vectorizer.fit\_transform(corpus) #(4, 13)  print(vectorizer.get\_feature\_name())  print(X.toarray())  # Convert a new document to count representation  vectorizer.transform(['This is a new document']) |

得到一个非常稀疏的矩阵，每行表示一个Document，每一列表示一个Word。

这样的表示方法简单容易理解，而且编程也很容易实现，只需要取对应的索引就能够完成，已经可以解决相当一部分NLP的问题，但是仍然存在不足，即词向量与词向量之间都是相互独立的；我们都知道，词与词之间是有一定的联系的，我们无法通过这种词向量得知两个词在语义上是否相似，并且如果词表非常大的情况下，每个词都是茫茫0海中的一个1，这种高维稀疏的表示也有可能引发维度灾难。为了解决上述问题，就有了词向量的第二种表示方法。

# 分布式词向量编码

分布式编码（Distributed representation）最大的贡献就是让相关或者相似的词，在距离上更接近了。向量的距离可以用传统的欧氏距离来衡量，也可以用余弦相似度来衡量。Word2vec属于分布式词向量的一种。

将word映射到一个新的空间中，并以多维的连续实数向量进行表示叫做“Word Represention”或“Word Embedding”。自从21世纪以来，人们逐渐从原始的词向量稀疏表示法过渡到现在的低维空间中的密集表示。用稀疏表示法在解决实际问题时经常会遇到维数灾难，并且语义信息无法表示，无法揭示word之间的潜在联系。而采用低维空间表示法，不但解决了维数灾难问题，并且挖掘了word之间的关联属性，从而提高了向量语义上的准确度。

简单说，分布式词向量编码具有以下特点：（1）将vector每一个元素由整形改为浮点型，变为整个实数范围的表示；（2）转化为低维度的连续值，也就是稠密向量。将原来稀疏的巨大维度压缩嵌入到一个更小维度的空间。并且其中意思相近的词将被映射到向量空间中相近的位置。

# Word2Vec

Word2vec是在Google Tomas Mikolov在2013年提出来的，作用同样是将word转换成vector表达，但是这样的vector是低维度的实数值。学习到的这个vector就是Embedding/词嵌入向量。

Word2Vec包括两种模型，CBOW与Skip-gram。前者利用单词上下文预测单词，后者利用单词预测上下文。

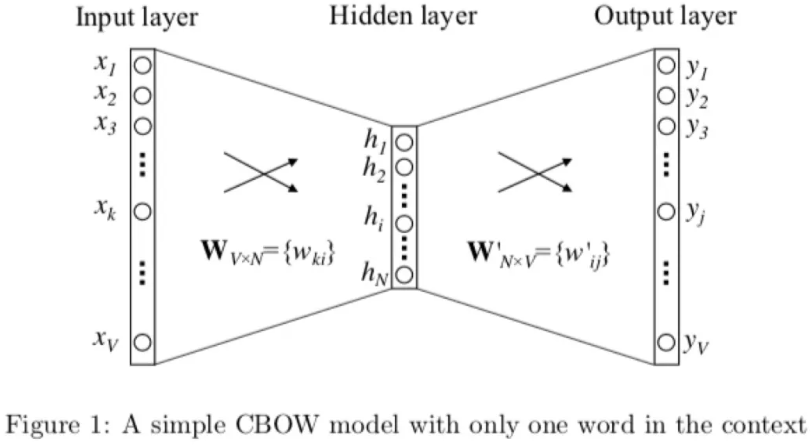
## CBOW模型

CBOW全称是Continuous Bag Of Words, 是指利用上下文单词来预测一个词。这个上下文可以只有上文/下文，也可以上下文都有。具体是多长的上下文，是人为指定的，也称为窗口大小。

### One-word Context

先来看最简单的情况，s目标是：给出当前上下文（只有一个词），求出下一个词的概率分布。也就是下一个词是字典中的某个词的概率，概率有大有小，最大的那个就是模型预测出来的下一个词。为了保证这个词真的是语料中下一个词，我们需要调整概率分布使得其概率最大，这个过程就是学习训练的过程。

此时的网络结构如下图所示。V是词典中word的数量，N表示词向量的维度/嵌入的维度。

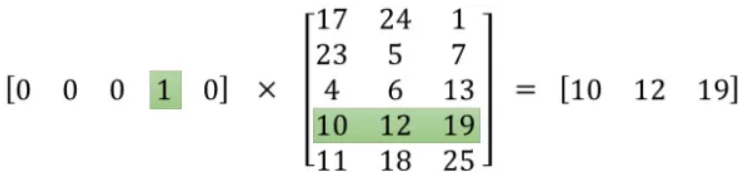


* 输入只有一个单词，用one hot的形式表示，V维度
* 中间隐藏层维度为N，也就是嵌入的维度，没有激活函数。
* 输出层使用softmax激活函数，维度为V

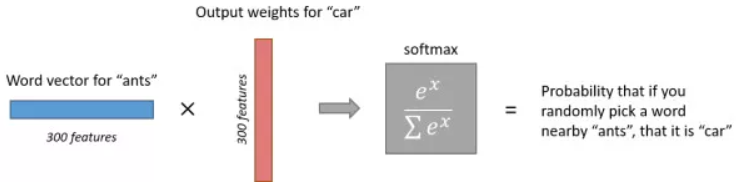
这是最基本的模型，使用softmax计算复杂度为O(V)，可以使用层次softmax或Negative Sampling进行优化。此处不再展开，后续会专门来写。

因此，输入，输出都是V维向量，中间一个隐藏层。而且，输入是one hot形式的V维度向量，输出使用softmax激活函数，隐藏层没有激活函数。

鉴于输入是one hot形式的，只有一个非零值且为1，所以相当于是在input -> hidden layer权重矩阵中选择第k行，k对应输入one hot中不为零的那一项，如下图所示。此时的隐藏层矩阵h，称为输入单词的input vector，维度为N。



hidden->output layer矩阵是另外一个矩阵，它完成从隐藏层到输出层的映射。W’中每一列依次与h相乘，就对应着输出层中每一个神经元的值，而输出层的每一个神经元对应着一个单词。如下图所示。



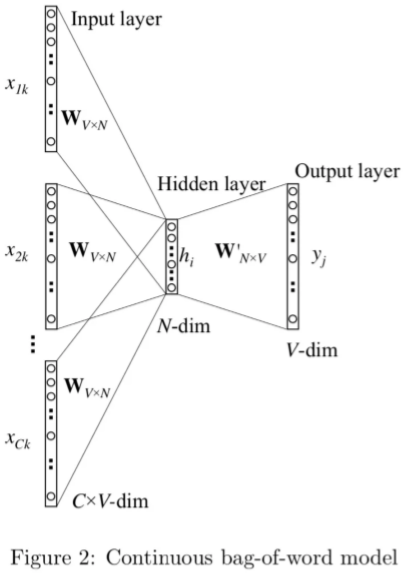
所以W'每一列也对应着字典中一个单词，我们称该列为对应单词的**output vector**，维度为N。

input vector就是对应词的嵌入向量。也就是说，最终想要的就是input -> hidden layer的权重矩阵，矩阵shape=(V,N)，每一行都是一个N维向量，就是对应位置的词向量。

### Multi-word Context

上下文只有一个单词的情况清楚后，上下文有多个单词的情况也就非常简单了。输入由原来的一个one hot vector，变成如今的多个one hot vector。他们使用同一个input -> hidden layer权重矩阵，选出其中对应的行。然后对其取平均值作为中间隐藏层的输出。其余的和单一上下文的情况是相同的。

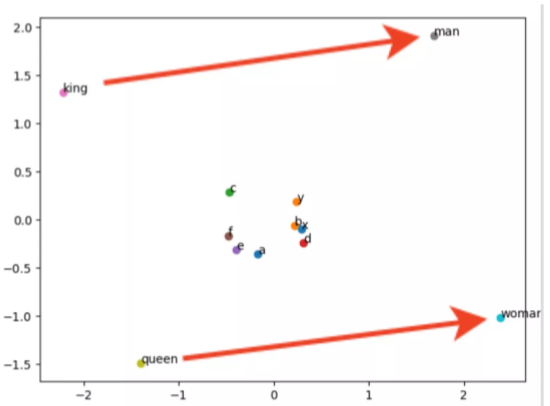
如下图所示：



采用tensorflow实现简单的One-word Context的代码见：

|  |
| --- |
| import numpy as np  from sklearn.preprocessing import OneHotEncoder  from tensorflow import keras  from tensorflow.keras.layers import Input, Dense  from tensorflow.keras.models import Model  import matplotlib.pyplot as plt  from sklearn.decomposition import PCA  """  采用tensorflow实现cbow的one word context版本  """  """  1. 准备训练数据  """  # Context|Target  corpus\_king\_queen\_symbol = ['king|a', 'queen|a', 'king|b', 'queen|b', 'king|c', 'queen|c', 'king|x',  'queen|y', 'man|d', 'woman|d', 'man|e', 'woman|e', 'man|f', 'woman|f',  'man|x', 'woman|y']  train\_data = [sample.split('|')[0] for sample in corpus\_king\_queen\_symbol]  train\_label = [sample.split('|')[1] for sample in corpus\_king\_queen\_symbol]  vocabulary = (list(set(train\_data) | set(train\_label)))  vocabulary.sort()  one\_hot\_encoder = OneHotEncoder()  one\_hot\_encoder.fit(np.reshape(vocabulary, (-1, 1)))  X = one\_hot\_encoder.transform(np.reshape(train\_data, (-1, 1))).toarray()  y = one\_hot\_encoder.transform(np.reshape(train\_label, (-1, 1))).toarray()  """  2. 构建模型  输入是X，y  """  N = 5  V = len(vocabulary)  inputs = Input(shape=(V, ))  x = Dense(N, activation='linear', use\_bias=False)(inputs)  predictions = Dense(V, activation='softmax', use\_bias=False)(x)  model = Model(inputs=inputs, outputs=predictions)  model.summary()  """  3. 训练模型  """  model.compile(optimizer=keras.optimizers.Adagrad(0.07), loss='categorical\_crossentropy', metrics=['accuracy', 'mse'])  model.fit(X, y, batch\_size=1, epochs=1000)  """  4. 验证/可视化结果  """  weights = model.get\_weights()  embeddings = np.array(weights[0])  assert (embeddings.shape == (V, N))  word\_vec = dict((word, vector) for word, vector in zip(vocabulary, embeddings))  pca = PCA(n\_components=2)  X\_pca = pca.fit\_transform(embeddings)  print(X\_pca)  fig, ax = plt.subplots()  for i in range(len(X\_pca)):  team = X\_pca[i]  ax.scatter(team[0], team[1])  ax.annotate(vocabulary[i], (team[0], team[1]))  plt.show() |

训练之后，可以看到经典的(king - man) + woman = queen如下图所示：

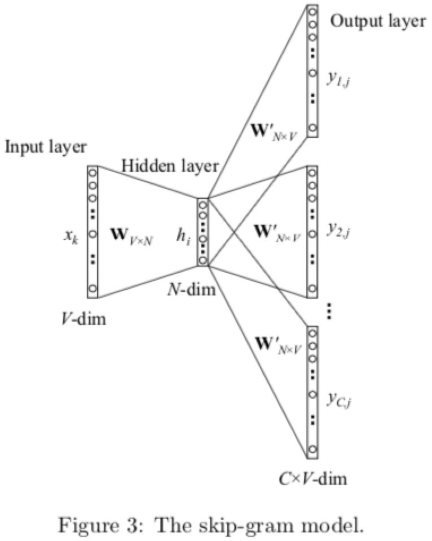


## Skip-gram

### Skip-gram理论

Word2Vec是为了得到word的低维度实数向量表示，为此，需要根据原始语料中构造一个假的分类问题。CBOW是给出上下文Context，然后预测中心（目标）词；Skip-gram是给出中心（目标）词，预测上下文。两者都有个共同点就是，input->hidden weights就是Embedding权重矩阵，每一行对应一个单词的embedding vector / word2vec /低维度实数向量表示。

语言概率模型是指给出一个句子，然后判断这个句子是人话的概率。即每个单词组合在 一起是正确的概率。Skip-gram是指根据一个当前词预测多个上下文单词，是一对多的关系。如下图所示：



模型其实非常简单，需要注意的主要有以下几点：

1. 输入是当前单词的one hot向量。跟CBOW一样，input -> hidden的weights就是Embedding矩阵，V行N列，每一行对应字典中一个单词的嵌入向量，在Word2Vec中也叫做输入单词的input vector；
2. 中间隐藏层没有激活函数，只是线性投影；
3. 中间隐藏层到输出层共享权重矩阵W’ shape=(N,V)，注意与input -> hidden权重矩阵是两个不同的矩阵。W’中每一列也对应一个单词，被称为该单词的output vector；
4. 输出层将input vector和output vector进行点乘，然后使用softmax进行归一化，就得到了在当前单词输入下，输出是对应单词的概率；

注意：上图并不是网络结构！Skip-gram在训练的时候，输入输出都只是一个单词，和CBOW中one context的网络结构是相同的。

对于同一个中心词，具有多个上下文词，模型的目标函数是将整体的交叉熵降到最低，模型会找到一个中间的平衡点，使得loss最低。并不会因为输入和输出不同造成网络混乱。

训练完之后，输入中心词，输出单词跟语料中每个单词的统计次数有关。虽然同一个中心词对应不同的上下文，但是有的上下文单词出现的多，有的出现的少，它们会影响loss的计算。为了降低loss，模型倾向于预测出现较多的上下文单词。其实并不关心预测哪个单词，关心的只是使得loss最小的训练后模型的input -> hidden权重。

### 训练样本生成

下面用一个例子说明下上述整个过程：

输入语料可以是一句话，或者一段话，或者一篇文章。假设语料为一句话the quick brown fox jumped over the lazy dog 设window为1，即只考虑上下各一个单词，那么训练样本则是:

|  |
| --- |
| quick -> (the, brown)  brown -> (quick, fox)  fox -> (brown, jumped)  ... |

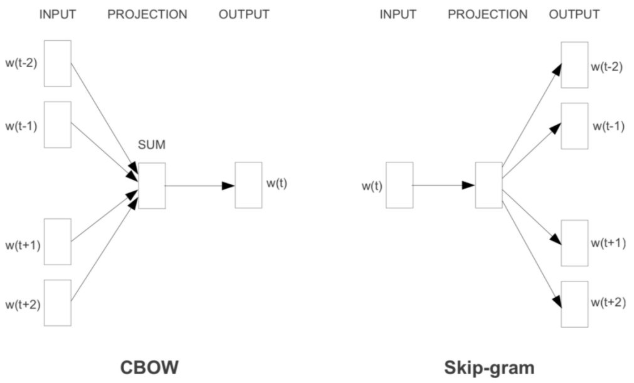
Skip-gram训练样本是一对一对的，所以实际生成的训练样本是这样的：

|  |
| --- |
| // 这才是实际的训练样本  (quick, the)  (quick, brown)  (brown, quick)  (brown, fox)  (fox, brown)  (fox, jumped)  。。。 |

## CBOW vs Skip-gram

CBOW与Skip-gram的区别：

1. CBOW是利用上下文预测中心词，Skip-gram是利用中心词预测上下文
2. Skip-gram效果比CBOW好。
3. Skip-gram训练时间长，但是对低频词(生僻词)效果好；CBOW训练时间短，对低频词效果比较差。



两者结构对比如上图所示。对于上面的差异可以按照如下解释进行理解：

CBOW是利用上下文预测中心词，训练过程是从输出的中心词的loss来学习上下文的词向量。V个中心词，对应V个训练样本，一共学习V次就结束了。训练复杂度O(V)，训练时间较快。而且上下文词向量是取得平均值，一视同仁的对待，那么低频词训练的少，而且也没有特殊处理，当然效果不好。

Skip-gram利用中心词预测上下文，假设我们考虑前后共K个上下文，那么每一个上下文都会修正一次中心词的词向量表达，训练复杂度是O(KV)训练时间会变长。但是低频词也有多个上下文，相比于CBOW，其词向量会被多次修正，自然效果也就好一些。

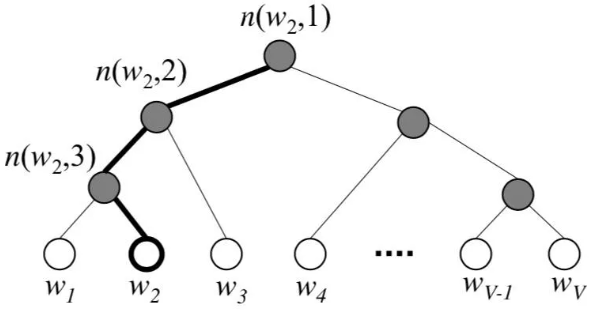
在通俗一点来讲，CBOW是一个老师多个学生，每个学生平等对待，学生能学多少，看你上了多少次老师的课（作为上下文被修正了多少次）。Skip-gram是多个老师一个学生，即使这个学生出现的次数很少，但是每次上课都是多个老师在教他，自然就学的多，从整个训练的角度来看，自然花费的时间也就长，毕竟每个学生都要被多个老师教一遍。

注：Skip-gram的网络结构和CBOW中one context是一样的，只是训练样本由(context, target)换成了（target，context）

## Word2Vec优化策略

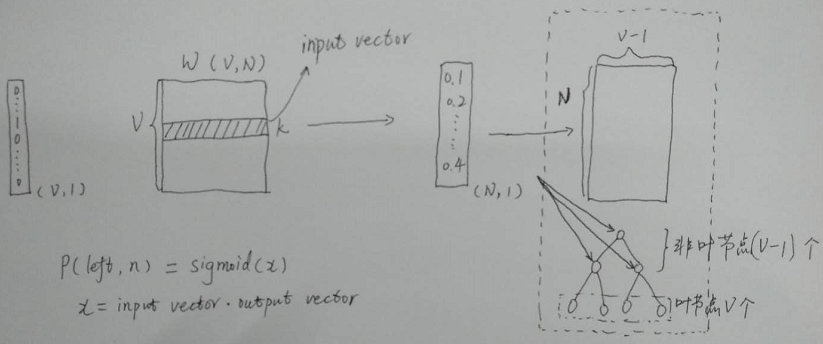
### Hierarchical Softmax

原始的Word2Vec使用softmax得到最终的词汇概率分布，词汇表往往包含上百万个单词，如果针对输出中每一个单词都要用softmax计算概率的话，计算量是非常大的。解决办法之一就是Hierarchical Softmax。相比于原始的Softmax直接计算每个单词的概率，Hierarchical Softmax使用一颗二叉树来得到每个单词的概率。被验证的效果最好的二叉树类型就是哈夫曼树：



哈夫曼树中有V-1个中间节点，V个叶节点。叶节点与单词表中V个单词一一对应。首先根据单词出现的频率构造一颗霍夫曼树，出现频率高的单词霍夫曼编码就短，更加靠近根节点。

原来的Word2Vec模型结构会被改变，隐藏层后直接和哈夫曼树中每一个非叶节点相连，如下图所示（相当于输出层中只有V-1个神经元节点）。然后在每一个非叶节点上计算二分概率（也就是用Sigmoid函数进行激活），这个概率是指从当前节点随机游走的概率，可以任意指定是向左游走的概率，还是向右游走的概率。从根节点到目标单词的路径是唯一的，将中间非叶节点的游走概率相乘就得到了最终目标单词的概率。



这样只用计算树深度个输出节点的概率就可以得到目标单词的概率。哈夫曼树的深度基本是logV，所以此时的计算复杂度就降为了O(logV)。另外，高频词非常接近树根，其所需要的计算次数将进一步减少，这也是使用哈夫曼树的一个优点。此时的目标函数为：

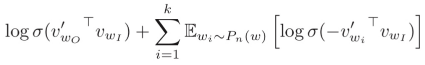


L(w)表示树深度；n(w,j)表示从根节点到目标单词w的路径上第j个节点；ch(n)表示节点n的孩子节点。中间的尖括号表示是否成立的判断，结果无非是+1，或-1。注意：Sigmoid(-x) = 1 - Sigmoid(x)。Vwi表示输入单词的input vector，Vn’表示霍夫曼树中间节点的output vector。

### Negative Sampling

Negative Sampling（简称NEG）是NCE（Noise Contrastive Estimation）的简化版本，目的是提高训练速度并改善所得词向量的质量。与Hierarchical Softmax相比，NEG不再采用复杂的哈夫曼树，而是利用相对简单的随机负采样，能大幅提升性能，因而可以作为Hierarchical Softmax的一种替代。

针对一个样本(WI, W)，Negative Sampling的目标函数如下所示：



现在的目标就是利用logistic regression，从带有噪声（负样本）的目标中找到正样本（Wo），其中K是负样本采样的数量。作者指出在小训练集上，k取5-20比较合适；大训练集上k取2-5即可。和SGD的思想非常像，不再是利用所有的负样本进行参数的更新，而是只利用负采样出来的K个来进行loss的计算和参数的更新。只不过SGD每次只用一个样本，而不是K个。

负样本采样服从分布Pn(W)，经过试验发现unigram分布效果最好，如下：

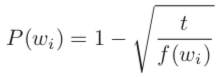


其中f(wi)表示单词wi在语料中出现的频率。3/4这个值是通过实验发现的经验值。

### Subsampling of Frequent Wrods

大语料集中，像the a in之类的单词出现频率非常高几乎是很多单词的上下文，造成其携带的信息非常少。对这些单词进行下采样不仅可以加快训练速度还可以提高低频词训练词向量的质量。

为了平衡高频与低频词，对训练集中的单词按照下述公式决定是否保留该单词：



其中f(wi)是单词wi出现的频率，t凭经验值取10的-5次方。该公式保证出现频率超过t的单词将被下采样，并且不会影响原有的单词的频率相对大小（rank）。先生成（target， context），比如（love，[ I,China]），然后依次遍历他们，小于P(wi)的训练样本将被从训练样本中去掉。

### 词组Phrases

词组并不能简单的将单词分开解释，而是应该看做一个整体，例如“New York”。把这样的词组看成是一个整体，用特殊的符号代替不会过多的增加词汇表的大小，但效果是很显著的。

# 实践

见github：<https://github.com/jpegbert/NLP_Coding/tree/master/word2vec>

代码包含tensorflow和pytorch，分别实现了cbow和skip-gram的Hierarchical Softmax和Negative Sampling四种版本，还包含一个简单的word2vec实验。

参考：

<https://mp.weixin.qq.com/s/6lxL80a3Yih9dTFDGJuHaw>

<https://mp.weixin.qq.com/s/98oNzdDtfEOqDql0LPibJw>

<https://mp.weixin.qq.com/s/BqZ0jJcCrekod31TPXFOSQ>