# Word2Vec

我看了一些Word2Vec的一些相关文章，主要分为两类。一是有关于Word2Vec的发展，主要是以下4篇文章起到奠基性的作用

1. A Neural Probabilistic Language Model.2003 (NNLM)
2. Recurrent neural network based language model.2010 (RNNLM)
3. Distributed Representations of Words and Phrases and their Compositionality.2013 (Skip Gram Model and CBOW)
4. Efficient Estimation of Word Representations in Vector Space.2013 (Skip Gram Model and CBOW)

二是关于Word2Vec应用的一些文章，一些有趣的idea像是如下的文章

1. Linguistic Regularities in Continuous Space Word Representations.2013
2. Exploiting Similarities among Languages for Machine Translation.2013
3. Distributed Representations of Sentences and Documents.2014

## Development

### NNLM: A Neural Probabilistic Language Model.2003

这篇文章主要解决在此之前的自然语言模型是统计语言模型和基于统计语言模型n-gram模型的维度灾难问题。

statistical language model:

$$\hat{P}(w_1^T) = \prod_{t=1}^{T} \hat{P}(w_t | w_1^{t-1}),$$

where $w_t$ is the $t$-th word, and writing sub-sequence $w_i^j = (w_i, w_{i+1}, \cdots, w_{j-1}, w_j)$.

N-gram model:

$$\hat{P}(w_t | w_1^{t-1}) \approx \hat{P}(w_t | w_{t-n+1}^{t-1}).$$

统计语言模型的基本想法就是对于一句话，在给定前几个词的情况下，统计出现下一个词的概率。这样一句话的出现概率就是第一个词出现的概率$P(W_1)$乘上在第一个词给定的情况下出现第二个词的概率$P(W_2|W_1)$, 依此类推，一句话的概率就是上图第一行的联合条件概率乘积。

N-gram模型就是假设一个词出现的概率只考察前后该词前后n个词，以此来降低复杂度。

这些模型的问题就是复杂度非常高，例如：

*curse of dimensionality*. It is particularly obvious in the case when one wants to model the joint distribution between many discrete random variables (such as words in a sentence, or discrete attributes in a data-mining task). For example, if one wants to model the joint distribution of 10 consecutive words in a natural language with a vocabulary $V$ of size 100,000, there are potentially $100\,000^{10} - 1 = 10^{50} - 1$ free parameters. When modeling continuous variables, we obtain gen-

上图的**free parameters**就是指前面语言模型的各个概率P. 统计语言模型就是要统计所有文本将所有概率P确定下来。

文章的作者要解决这一问题，采用distributed vectors来表示每一个词，一句话的上下文语境用训练好的vector的值和网络参数来表达。如下：

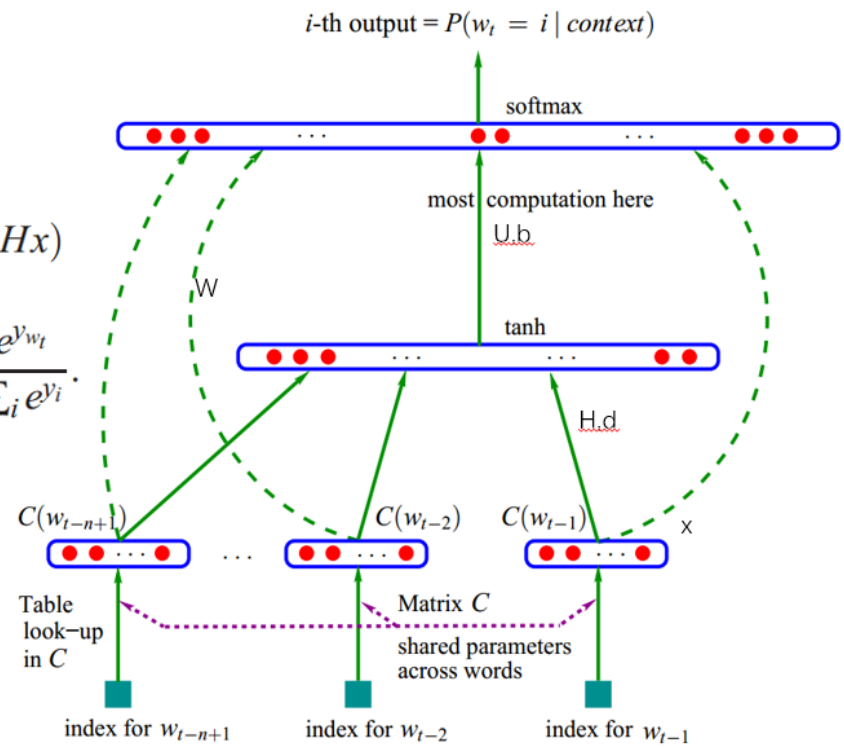
1. associate with each word in the vocabulary a distributed *word feature vector* (a real-valued vector in $\mathbb{R}^m$),
2. express the joint *probability function* of word sequences in terms of the feature vectors of these words in the sequence, and
3. learn simultaneously the *word feature vectors* and the parameters of that *probability function*.

作者给出的网络结构如下：

NNLM

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

$$\hat{P}(w_t|w_{t-1},\cdots w_{t-n+1}) = \frac{e^{y_{w_t}}}{\sum_i e^{y_i}}.$$




如图，首先有一个全局的矩阵C,通过C将一个词w转换为向量C(w)。然后用这些向量作为一个三层神经网络的输入，最终训练出参数和所有词向量的矩阵C。

作者首次使用神经网络来表示自然语言模型，大大降低了计算复杂度（复杂度在后面统一比较）。作者实际上已经使用了词向量但是没有发现这些向量蕴含的语义和语法内容，只是用它来作为自然语言模型的工具。

## RNNLM:Recurrent neural network based language model.2010

NNLM有一个问题就是对于一个词，训练的时候它的上下文只取该词前面的n个词，如果n过大，训练复杂度增大；n过小，覆盖的上下文不够。本文作者就利用RNN的特性来解决这一问题。

RNN:

$$x(t) = w(t) + s(t-1)$$

$$s_j(t) = f\left(\sum_i x_i(t)u_{ji}\right)$$

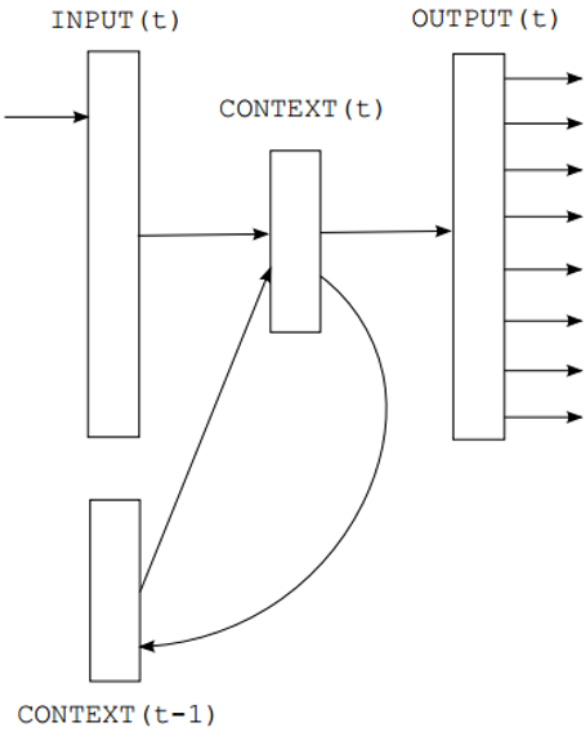
$$y_k(t) = g\left(\sum_j s_j(t)v_{kj}\right)$$

where $f(z)$ is sigmoid activation function:

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

and $g(z)$ is softmax function:

$$g(z_m) = \frac{e^{z_m}}{\sum_k e^{z_k}}$$



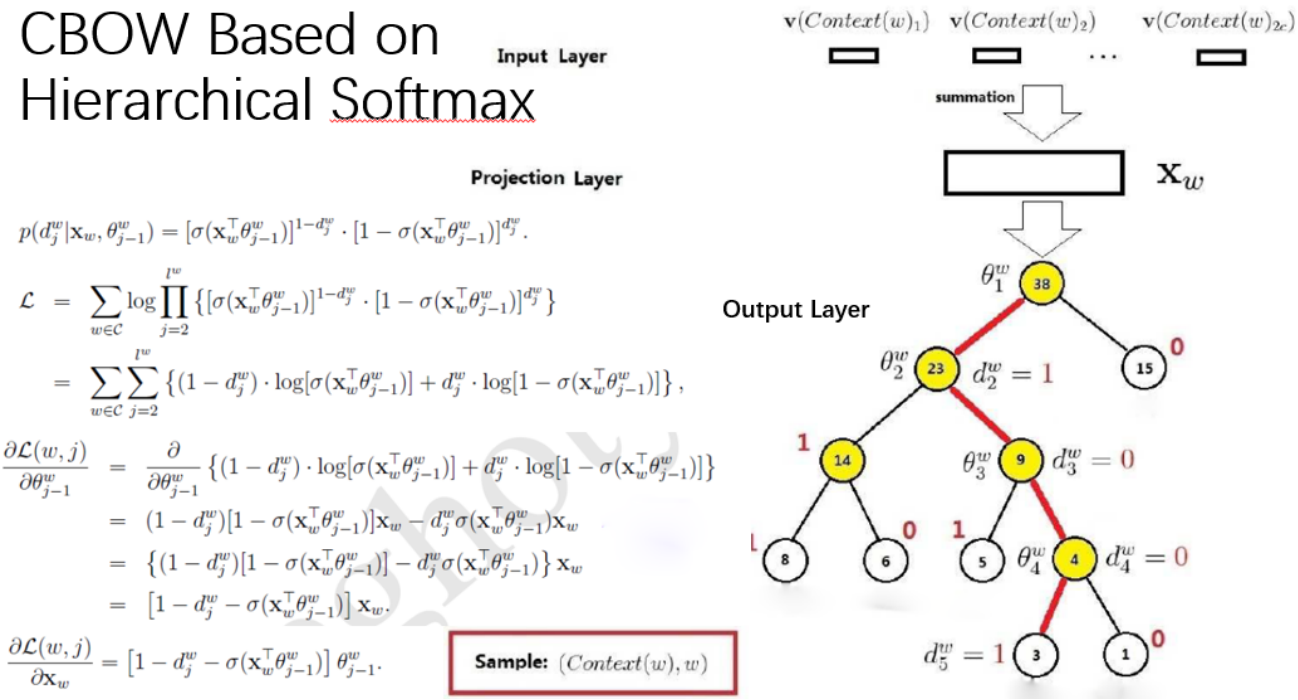

如上图，RNNLM中将隐藏层视为context层，每一次迭代中，输入由新的输入和前一次的context层共同组成。这样上下文的宽度n不用很大，也可以利用到距离当前词足够远的词带来的影响。

但是本文作者同样是没有注意到词向量蕴含的语义和语法内容

## Skip Gram Model and CBOW

“Distributed Representations of Words and Phrases and their Compositionality”和“Efficient Estimation of Word Representations in Vector Space”两篇文章提出了Skip Gram Model and CBOW两个模型。

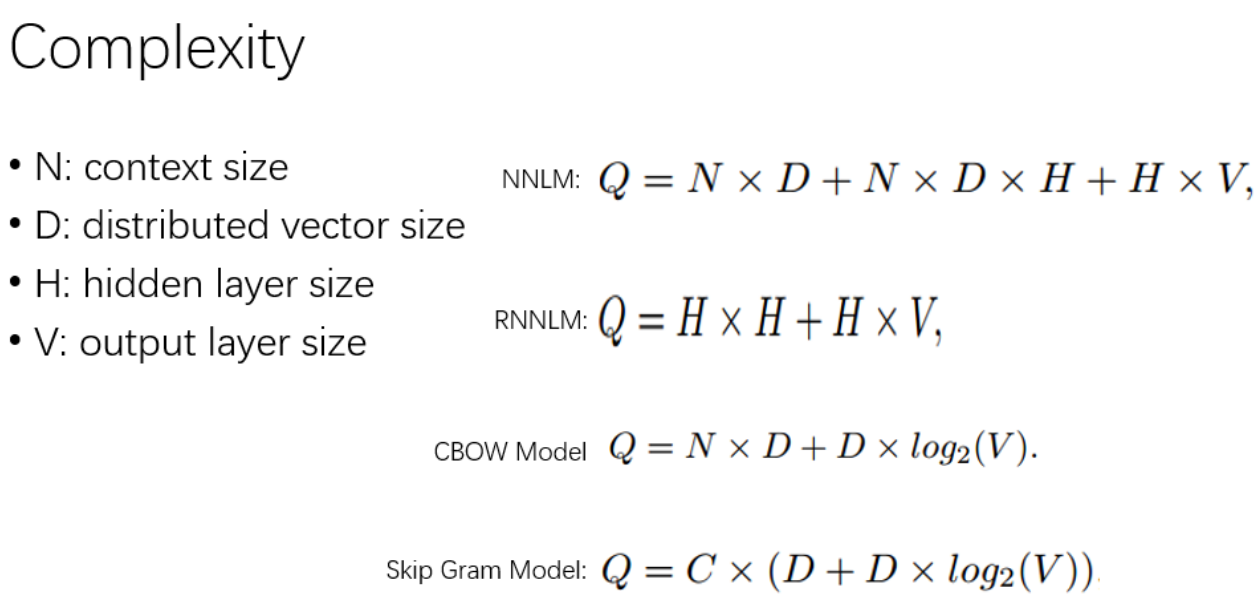
CBOW模型：




CBOW模型相比于前两个模型，去掉了隐藏层。输出不是线性的softmax，而是基于huffman树的softmax，这样首先使得输出层复杂度由N降到log(N)，其次很好的应用了huffman树的聚类效果。

Skip Gram Model与CBOW类似，只是CBOW是由当前词的上下文来推导出当前词汇，而Skip Gram Model则是反过来。

这两个模型首先是降低了计算复杂度，如下图比较：




按照文章的说法

> This makes the training extremely efficient: an optimized single-machine implementation can train on more than 100 billion words in one day

此外最重要的是，作者首次察觉到词向量蕴含了丰富的语法和语义的含义。比如

> vec(“Madrid”) - vec(“Spain”) + vec(“France”) is closer to vec(“Paris”) .

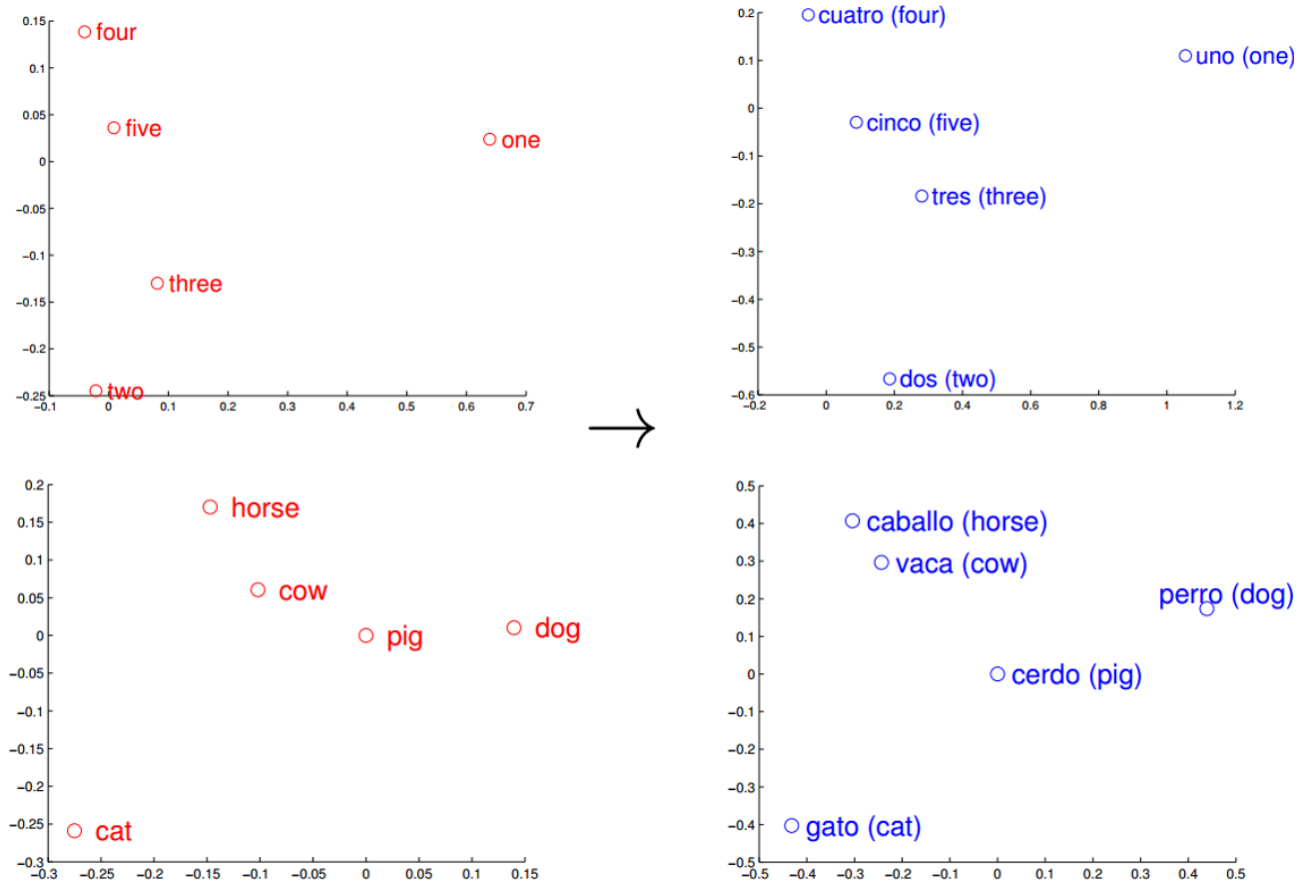
文章中用了大量篇幅和实现表现词向量的良好特性

# Application

然后还有些应用词向量的文章。

## Exploiting Similarities among Languages for Machine Translation

文章的想法非常简单。在一种语言中一个词的词向量为$X_i$，另一种语言中其对应的词的词向量为$Z_i$,作者做了一些实验发现这写词在各自的语言空间中的相对位置关系有一定的相似性，如下图：



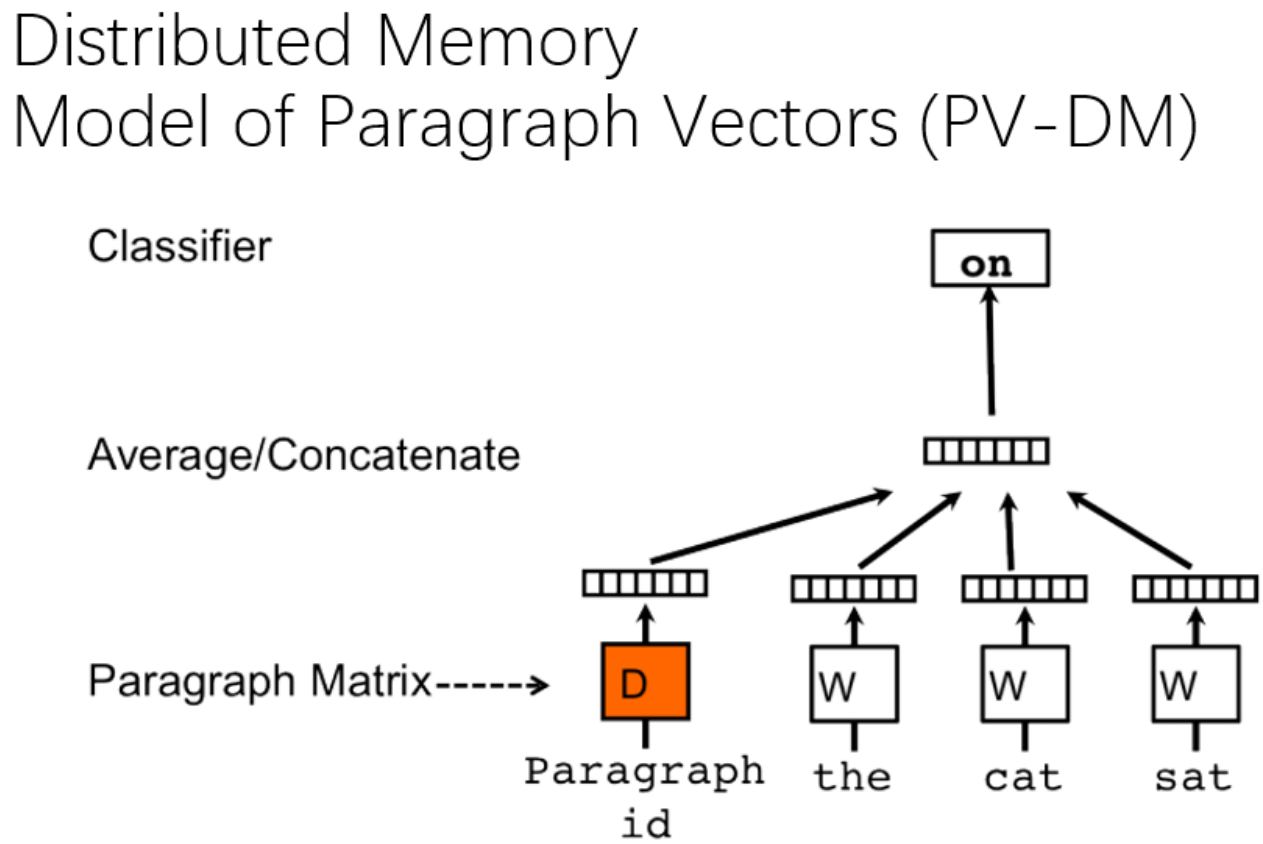

所以作者认为，只要训练一个转移矩阵W，使得$X_i$经过变换$WX_i$转换到与$Z_i$接近的位置，再转换成对应的词，这样就完成了翻译，训练目标如下图：

$$\min_W \sum_{i=1}^{n} \|W x_i - z_i\|^2$$

作者想法很新颖，但是模型结构非常简单，训练的过程中只考虑到词，没有融入整句话的语义概念。翻译的效果比不上之前的翻译模型，只是开阔了一种新的思路。

### Distributed Representations of Sentences and Documents

这一篇文章进一步扩展词向量，提出一种方法来训练句子和文章的向量。




如图，作者的想法也非常简单，在原有的CBOW模型上，作者在输入层额外添加了一个paragraph vector。这个向量是由段落决定的，文章的每一段对应一个向量，在训练过程中，同一段落内部使用同一个向量。这一个向量就代表着该段落的context and semantic.

这是一个非常简单但是巧妙的想法，效果也相当不错。作者做了一个实验来来评估模型，实验是分析一段评论是positive or negative，数据集是Stanford Sentiment Treebank Dataset。实验结果如下，错误率要低于之前的模型。

| Model | Error rate (Positive/ Negative) | Error rate (Fine-grained) |
|---|---|---|
| Naïve Bayes (Socher et al., 2013b) | 18.2 % | 59.0% |
| SVMs (Socher et al., 2013b) | 20.6% | 59.3% |
| Bigram Naïve Bayes (Socher et al., 2013b) | 16.9% | 58.1% |
| Word Vector Averaging (Socher et al., 2013b) | 19.9% | 67.3% |
| Recursive Neural Network (Socher et al., 2013b) | 17.6% | 56.8% |
| Matrix Vector-RNN (Socher et al., 2013b) | 17.1% | 55.6% |
| Recursive Neural Tensor Network (Socher et al., 2013b) | 14.6% | 54.3% |
| Paragraph Vector | **12.2%** | **51.3%** |

这一篇训练sentence vector还可以应用到前一篇翻译的工作中，可以弥补其没有融入整句话的语义概念的不足之处。

## 附：CBOW核心代码

这是目前工作中用到的一段CBOW核心代码，参数还没有调好。不过实现过程中，对于论文中写的比较概括模糊的地方有了比较清晰的认识。

```python
# -*- coding: utf-8 -*-
"""
Created on Tue Jan 10 17:07:40 2017

@author: duocai
"""

import numpy as np
import json
import math
import datetime
import random
import threading

#控制内积范围，超过范围的再加sigmoid很小，可以丢弃
MAX_EXP = 8

# 向量维度
layer_size = 100
# 单元location or word..个数
unit_num = 182968
# 单元向量表
vec_table = np.random.uniform(-1,1,layer_size*unit_num).reshape(unit_num,layer_size)
# 参数表，参数个数即内部节点个数为叶节点个数（unit_num）-1，初始化为正态分布取值
para_table =  np.random.normal(0.0,1,layer_size*(unit_num-1)).reshape(unit_num-1,layer_size)
# 所有用户，相当于所有文章
users = json.load(open('../user_new.json'))
# 所有地点，相当于所有单词
locats = json.load(open('../location_tree.v3.json'))
if len(locats.keys()) > unit_num:
    print('地点超出范围')
# window size
window = 5
# 学习率
start_alpha = 0.03
# 开始时间
start_time = datetime.datetime.now()

def sigmoid(x):
    return 1/(1 + math.exp(-x))

def show_message():
    print('vector of location 0:',vec_table[0])

# 训练index指定的用户
def train(index):
    global layer_size
    global window
    global MAX_EXP
    global start_alpha
    global start_time

    sent = users[index]['locations']#文章，用户访问地点序列
```

```
    sent_len = len(sent)
    neul = np.zeros(layer_size)#隐藏层
    neule = np.zeros(layer_size)# 记录隐藏层累积变化量
    for pos in range(sent_len):
        loc_id = sent[0] # 地点id string
        if (loc_id not in locats):#不存在该地点则忽略
            continue
        # 计算context向量和
        num = random.randint(0,window)#随机起点，并不是严格从0开始
        start = num
        while start < 2 * window + 1 :
            cur_pos = pos - window + start
            #左右几个词不包含当前词
            # or 现在位置超过范围，则跳过
            if start == window or cur_pos < 0 or cur_pos >= sent_len:
                start+=1
                continue
            # 计算context向量和
            neul += vec_table[int(sent[cur_pos])]
            start+=1
        #利用霍夫曼树计算
        loc = locats[loc_id] # 当前地点
        points = loc['points']
        codes = loc['codes']
        codes_len = len(codes)
        for i in range(codes_len):
            # 计算内积
            dot = np.dot(neul,para_table[int(points[i])])
            ##内积不在范围内直接丢弃
            if dot > MAX_EXP or dot < -MAX_EXP:
                continue
            ##simoid
            dot = sigmoid(dot)
            # 偏导数的公用部分*学习率alpha
            g = (1 - int(codes[i]) - dot)*start_alpha

            # 反向更新参数
            # 先更新隐藏层
            neule += g*para_table[int(points[i])]
            # 后更新参数
            para_table[int(points[i])] += g*neul

        ## 将更新传递到词向量
        start = num
        while start < 2 * window + 1 :
            cur_pos = pos - window + start
            #左右几个词不包含当前词
            # or 现在位置超过范围，则跳过
            if start == window or cur_pos < 0 or cur_pos >= sent_len:
                start+=1
                continue
            # 更新词向量

            vec_table[int(sent[cur_pos])] += neule
```

```
            start+=1
    # 输出现在的时间
    if index%200 == 0:
        print('user: ', index, ',time: ', (datetime.datetime.now() - start_time).seconds, 's')
        show_message()

## 利用多线程训练
threads = []
for i in range(len(users)):
    threads.append(threading.Thread(target=train,args=(i,)))
## run
for t in threads:
    t.setDaemon(True)
    t.start()
t.join()#等待子线程结束
```