

Distributed Representations of Sentences and Documents

-- Quoc Le & Tomas Mikolov (2014)

Review of Le & Mikolov (2014) and introduction into Word2Vec

Introduction into Word2Vec

The last two years, the method and tool developed by Mikolov et al. (2013) for learning continuous word embeddings has gained a lot of traction. The model forms the basis for the study of Le & Mikolov (2014). In order to better understand the methods in that study, I believe it would be good to first present a brief overview of Word2vec. For this overview I gratefully make use of the excellent, in depth explanation of Word2Vec by Xin Rong (2014) (<http://arxiv.org/abs/1411.2738>). To have a more complete understanding of the model, please have a look at that paper.

Word2Vec attempts to associate words with points in space. The spatial distance between words then describes the relation (similarity) between these words. Words that are spatially close, are similar. Words are represented by continuous vectors over d dimensions. This example shows the relation between a number of words where each word is represented by a vector of two dimensions:

In [4]:

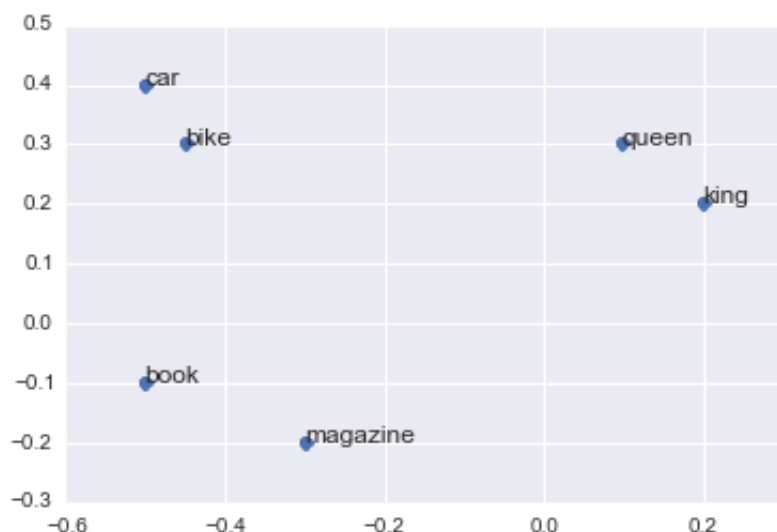
```
%matplotlib inline
```

In [5]:

```
import seaborn as sb
import numpy as np

words = ['queen', 'book', 'king', 'magazine', 'car', 'bike']
vectors = np.array([[0.1, 0.3], # queen
                    [-0.5, -0.1], # book
                    [0.2, 0.2], # king
                    [-0.3, -0.2], # magazine
                    [-0.5, 0.4], # car
                    [-0.45, 0.3]]) # bike

sb.plt.plot(vectors[:,0], vectors[:,1], 'o')
sb.plt.xlim(-0.6, 0.3)
sb.plt.ylim(-0.3, 0.5)
for word, x, y in zip(words, vectors[:,0], vectors[:,1]):
    sb.plt.annotate(word, (x, y), size=12)
```



The displacement vector (the vector between two vectors) describes the relation between two words. This makes it possible to compare displacement vectors to find pairs of words that have a similar relation to each other. A famous example given in the original paper is the following analogy relation: queen : king :: woman : man which should be read as queen relates to king in the same way as woman relates to man. In algebraic formulation: $v_{queen} - v_{king} = v_{woman} - v_{man}$. This technique of analogical reasoning can be applied to e.g. question answering.

Word2Vec learns continuous word embeddings from plain text. But how? The model assumes the *Distributional Hypothesis* that words are characterized by words they hang out with. We can use that idea to estimate the probability of two words occurring near each other, e.g. what is the probability of the following words, given *Cinderella*, i.e $P(w|Cinderella)$?

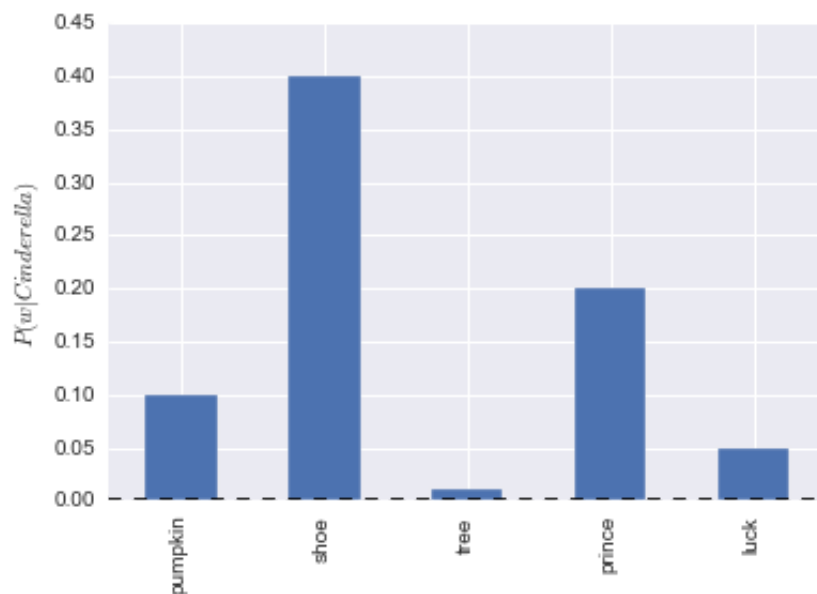
In [6]:

```
import pandas as pd

s = pd.Series([0.1, 0.4, 0.01, 0.2, 0.05],
              index=["pumpkin", "shoe", "tree", "prince", "luck"])
s.plot(kind='bar')
sb.plt.ylabel("$P(w|Cinderella)$")
```

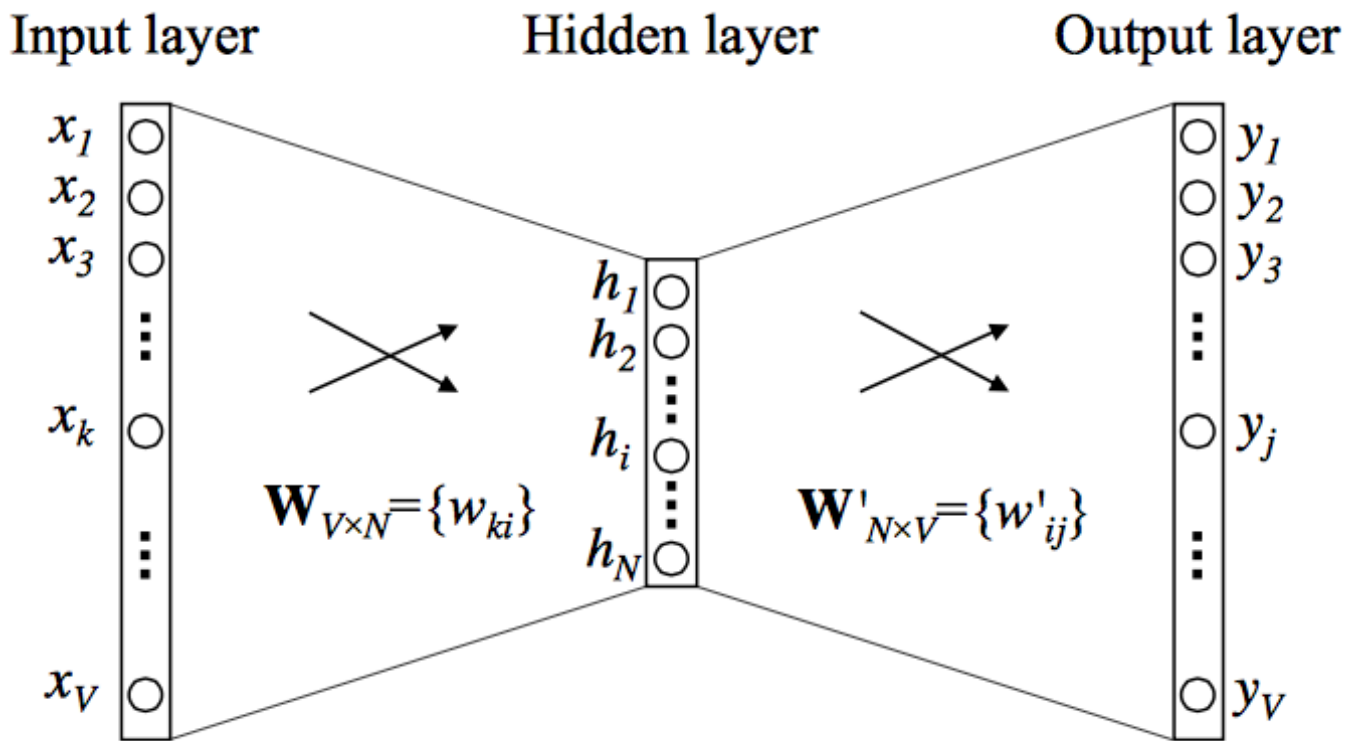
Out[6]:

<matplotlib.text.Text at 0x1136f8e90>



Softmax Regression

Word2Vec is a very simple neural network with a single hidden layer. Have a look at the following picture (taken from [Rong 2014 \(http://arxiv.org/abs/1411.2738\)](http://arxiv.org/abs/1411.2738)). We'll get into more details in a moment.



The model considers each word w_o in turn along with a given context C (e.g. $w_o \models \text{Cinderella}$ and $C \models \text{shoe}$). Now given this context, can we predict what w_o should be? This is essentially a multiclass classification where we have as many labels as our vocabulary size V . Using softmax regression, we can compute a probability distribution \hat{y} over the labels. The model attempts to minimize via Stochastic Gradient Descent the difference between the output distribution and the target distribution (which is a one-hot distribution which places all probability mass on the correct word). The difference between the two distribution is measured by the cross-entropy.

The neural network contains two matrices: W and W' of dimensions $V \times N$ and $N \times V$ respectively, where V is the vocabulary size and N the number of dimensions. Let's make this all a little more concrete with a small example. Say we have a corpus containing the following documents:

In [7]:

```
sentences = ['the king loves the queen', 'the queen loves the king',
             'the dwarf hates the king', 'the queen hates the dwarf',
             'the dwarf poisons the king', 'the dwarf poisons the queen']
```

We first transform these documents into bag-of-indices to enable easier computation:

In [8]:

```

from collections import defaultdict

def Vocabulary():
    dictionary = defaultdict()
    dictionary.default_factory = lambda: len(dictionary)
    return dictionary

def docs2bow(docs, dictionary):
    """Transforms a list of strings into a list of lists where
    each unique item is converted into a unique integer."""
    for doc in docs:
        yield [dictionary[word] for word in doc.split()]

```

In [9]:

```

vocabulary = Vocabulary()
sentences_bow = list(docs2bow(sentences, vocabulary))
sentences_bow

```

Out[9]:

```

[[0, 1, 2, 0, 3],
 [0, 3, 2, 0, 1],
 [0, 4, 5, 0, 1],
 [0, 3, 5, 0, 4],
 [0, 4, 6, 0, 1],
 [0, 4, 6, 0, 3]]

```

We now construct the two matrices W and W' :

In [10]:

```

import numpy as np

V, N = len(vocabulary), 3
WI = (np.random.random((V, N)) - 0.5) / N
WO = (np.random.random((N, V)) - 0.5) / V

```

Each row i in W corresponds to word i and each column j corresponds to the j th dimension.

In [11]:

```

print WI

```

```

[[-0.11953936  0.161062   -0.03209868]
 [ 0.00244538  0.11746195  0.12607279]
 [-0.14451005 -0.14540946  0.12184834]
 [-0.15411106 -0.04843555 -0.06943384]
 [ 0.04241556  0.01833369 -0.01999708]
 [ 0.09488057 -0.06968794  0.02873621]
 [-0.02498429  0.00360255  0.05691627]]

```

Notice that W' isn't simply the transpose of W but a different matrix:

In [12]:

print WO

```

[[-0.04019078  0.00870839 -0.00461927 -0.0255281  -0.00184819 -0.
017375
  0.04739917]
 [ 0.00962237  0.05808629 -0.02183692  0.07120648  0.0051645  0.
03994698
 -0.002765   ]
 [-0.03059303  0.05618066  0.029233    0.04137082  0.03396673 -0.
01705916
 -0.01613436]]

```

With the two matrices in place we continue with computing the posterior probability of an output word given some input word. Given an input word w_I , e.g. *dwarf* and its corresponding vector W_I , what is the probability that the output word w_O is *hates*? Using the dot product $W_I \cdot W_O'^T$ we compute the distance between the input word *dwarf* and the output word *hates*:

In [13]:

print np.dot(WI[vocabulary['dwarf']], WO.T[vocabulary['hates']])

0.000336538415207

Now using softmax regression, we can compute the posterior probability $P(w_O|w_I)$:

$$P(w_O|w_I) = y_i = \frac{\exp(W_I \cdot W_O'^T)}{\sum_{j=1}^V \exp(W_I \cdot W_j'^T)}$$

In [14]:

```

p_hates_dwarf = (np.exp(np.dot(WI[vocabulary['dwarf']], WO.T[vocabulary['hates']]),
                    sum(np.exp(np.dot(WI[vocabulary['dwarf']], WO.T[vocabulary[w]]))
                        for w in vocabulary))
print p_hates_dwarf

```

0.14291402521

Updating the hidden-to-output layer weights

Word2Vec attempts to associate words with points in space. These points in space are represented by the continuous embeddings of the words. All vectors are initialized as random points in space, so we need to *learn* better positions. The model does so by maximizing the equation above. The corresponding loss function which we try to minimize is $E = -\log P(w_O|w_I)$. First, let's focus on how to update the hidden-to-output layer weights. Say the target output word is *Cinderella*. Given the aforementioned one-hot target distribution t , the error can be computed as $t_j - y_j = e_j$, where t_j is 1 if w_j is the actual output word. So, the actual output word is *Cinderella* and we compute the posterior probability of $P(\text{pumpkin} | \text{tree})$, the error will be $0 - P(\text{pumpkin} | \text{tree})$, because *pumpkin* isn't the actual output word.

To obtain the gradient on the hidden-to-output weights, we compute $e_j \cdot h_i$ where h_i is a copy of the vector corresponding to the input word (only holds with a context of a single word). Finally, using stochastic gradient descent, with a learning rate ν we obtain the weight update equation for the hidden to output layer weights:

$$W_j'^{T(t)} = W_j'^{T(t-1)} - \nu \cdot e_j \cdot h_j$$

Assume the target word is *king* and the context or input word C is *queen*. Given this input word we compute for each word in the vocabulary the posterior probability $P(\text{word} \mid \text{queen})$. If the word is our target word, the error will be $1 - P(\text{word} \mid \text{queen})$; otherwise $0 - P(\text{word} \mid \text{queen})$. Finally, using stochastic gradient descent we update the hidden-to-output layer weights:

In [15]:

```
target_word = 'king'
input_word = 'queen'
learning_rate = 1.0

for word in vocabulary:
    p_word_queen = (np.exp(np.dot(WO.T[vocabulary[word]], WI[vocabulary[input_
                                sum(np.exp(np.dot(WO.T[vocabulary[w]], WI[vocabulary[input_
                                    for w in vocabulary))
    t = 1 if word == target_word else 0
    error = t - p_word_queen
    WO.T[vocabulary[word]] = (WO.T[vocabulary[word]] - learning_rate *
                              error * WI[vocabulary[input_word]])

print WO

[[-0.06243581  0.14095147 -0.02669861 -0.04758839 -0.0239239  -0.
03947435
  0.02543349]
 [ 0.00263098  0.09964895 -0.02877623  0.06427316 -0.00177368  0.
03300138
 -0.00966859]
 [-0.04061539  0.11576201  0.01928528  0.03143168  0.02402065 -0.
02701589
 -0.02603086]]
```

Updating the input-to-hidden layer weights

Now that we have a way to update the hidden-to-output layer weights, we concentrate on updating the input-to-hidden layer weights. We need to backpropagate the prediction errors to the input-to-hidden weights. We first compute EH which is an N dimensional vector representing the sum of the hidden-to-output vectors for each word in the vocabulary weighted by their prediction error:

$$\sum_{j=1}^V e_j \cdot W_{ij}' = EH_i$$

Again using the learning rate ν we update the weights using:

$$W_{wI}^{(t)} = W_{wI}^{(t-1)} - \nu \cdot EH$$

Let's see how that works in Python:

In [16]:

```
WI[vocabulary[input_word]] = WI[vocabulary[input_word]] - learning_rate * WO.s
```

If we now would recompute the probability of each word given the input word *queen*, we see that the probability of *king* given *queen* has gone up:

In [17]:

```
for word in vocabulary:
    p = (np.exp(np.dot(WO.T[vocabulary[word]], WI[vocabulary[input_word]])) /
         sum(np.exp(np.dot(WO.T[vocabulary[w]], WI[vocabulary[input_word]]))
              for w in vocabulary))
    print word, p
```

```
king 0.135793930736
dwarf 0.143640274055
queen 0.141911808054
poisons 0.144219025126
loves 0.14461048255
the 0.1457335424
hates 0.144090937079
```

Multi-word context

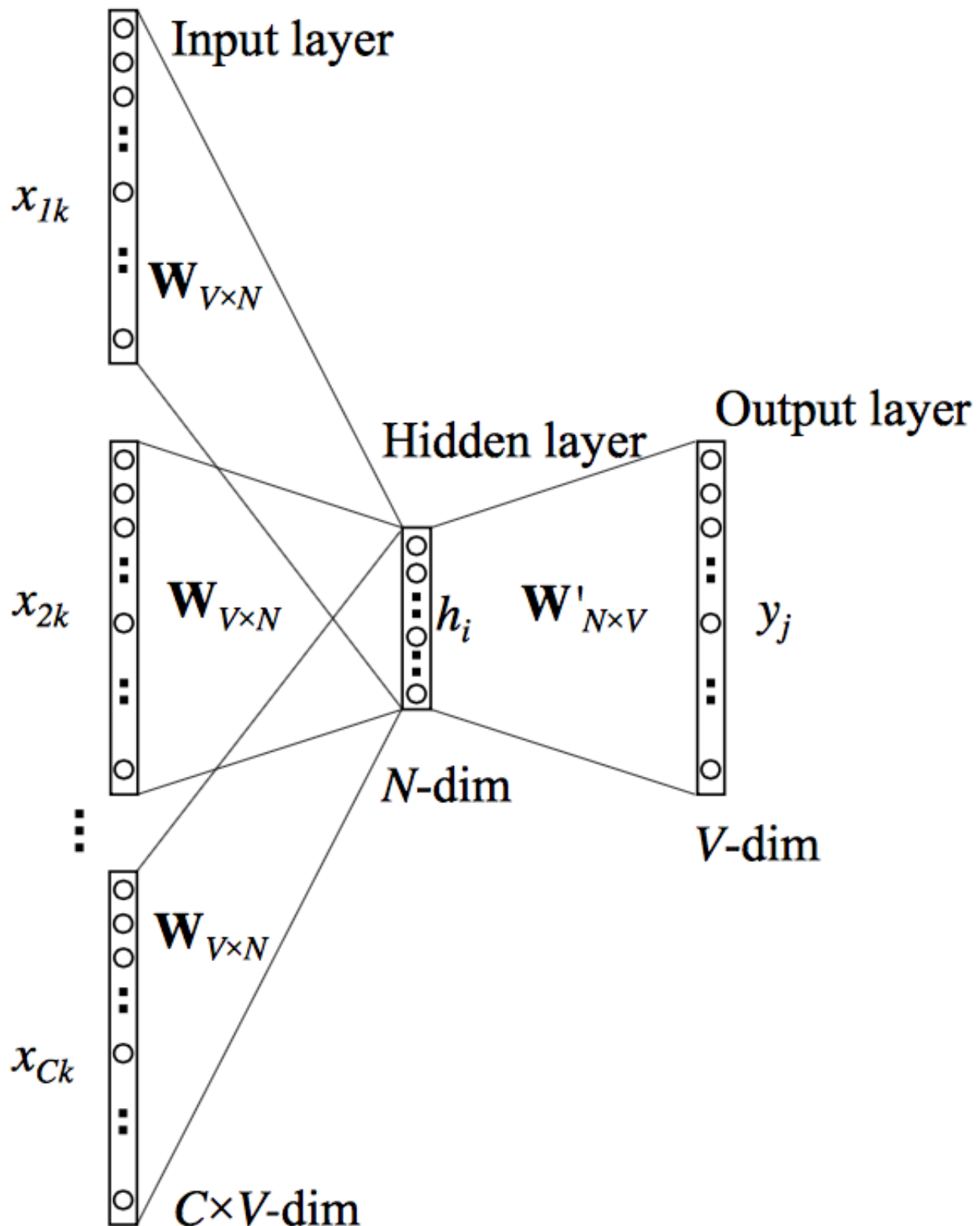


Figure taken from (Rong 2014)

The model described above is the CBOW architecture of Word2Vec. However, we assumed that the context C was only a single input word. This allowed us to simply copy the input vector to the hidden layer. If the context C comprises multiple words, instead of copying the input vector we take the mean of their input vectors as our hidden layer:

$$h = \frac{1}{C} (W_1 + W_2 + \dots + W_C)$$

The update functions remain the same except that for the update of the input vectors, we need to apply the update to each word in the context C :

$$W_{wI}^{(t)} = W_{wI}^{(t-1)} - \frac{1}{C} \cdot \nu \cdot EH$$

Let's see that in action. Again assume the target word is *king*. The context consists of two words: *queen* and *loves*.

In [18]:

```
target_word = 'king'
context = ['queen', 'loves']
```

We first take the average of the two context vectors:

In [19]:

```
h = (WI[vocabulary['queen']] + WI[vocabulary['loves']]) / 2
```

Then we apply the hidden-to-output layer update:

In [20]:

```
for word in vocabulary:
    p_word_context = (np.exp(np.dot(WO.T[vocabulary[word]], h)) /
                      sum(np.exp(np.dot(WO.T[vocabulary[w]], h)) for w in vocabulary))
    t = 1 if word == target_word else 0
    error = t - p_word_context
    WO.T[vocabulary[word]] = WO.T[vocabulary[word]] - learning_rate * error * t
print WO
```

```
[[-0.08162148  0.25512914 -0.04589425 -0.06655692 -0.04307783 -0.
05847402
  0.0063956 ]
 [-0.02294998  0.25188627 -0.0543705   0.03898171 -0.02731232  0.
00766842
 -0.03505252]
 [-0.04383296  0.13491036  0.01606604  0.02825053  0.02080841 -0.
03020226
 -0.02922364]]
```

Finally we update the vector of each input word in the context:

In [21]:

```
for input_word in context:
    WI[vocabulary[input_word]] = (WI[vocabulary[input_word]] - (1. / len(context) *
                                                                    learning_rate * WO.sum(1)))
```

In [22]:

```
h = (WI[vocabulary['queen']] + WI[vocabulary['loves']]) / 2
for word in vocabulary:
    p = (np.exp(np.dot(WO.T[vocabulary[word]], h)) /
         sum(np.exp(np.dot(WO.T[vocabulary[w]], h)) for w in vocabulary))
    print word, p
```

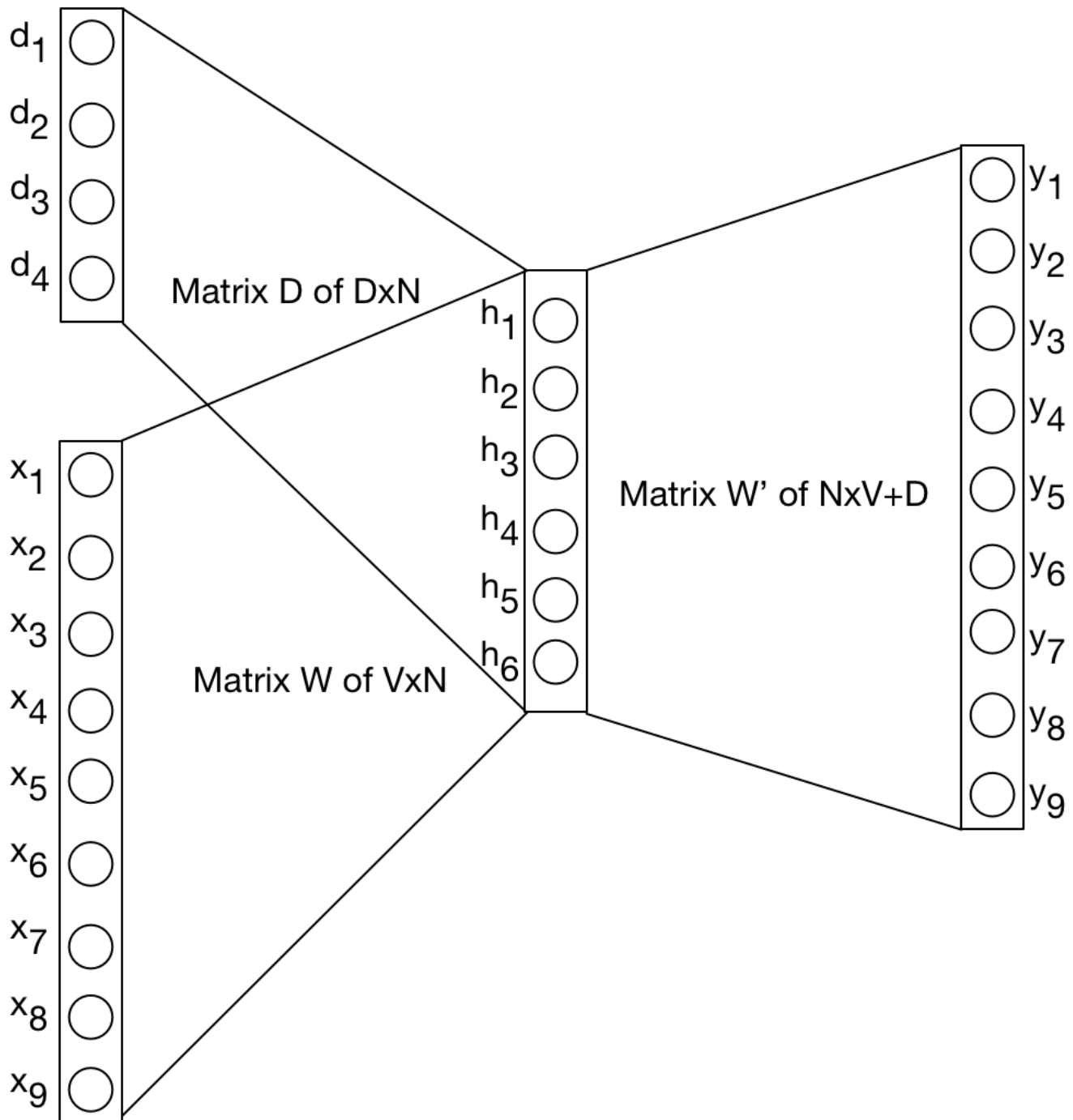
```
king 0.129518690596
dwarf 0.145150475243
queen 0.143019721408
poisons 0.145122173218
loves 0.146255939585
the 0.146300899102
hates 0.144632100847
```

Paragraph Vector

Hopefully we now have a better understanding of how Word2Vec learns continuous word embeddings from texts. Note that in order to efficiently apply this algorithm to real texts, we need some more tricks which I won't cover here. For now I would like to proceed with the Paragraph Vector described in Le & Mikolov (2014).

The Paragraph Vector model attempts to learn fixed-length continuous representations from variable-length pieces of text. These representations combine bag-of-words features with word semantics and can be used in all kinds of NLP applications.

If you read the paper, I think it will be clear by now that the Paragraph Vector is only a very small extension of the original model. Similar to the Word2Vec model, the Paragraph Vector model also attempts to predict the next word in a sentence. The only real difference, as the paper itself states, is with the computation of h . Where in the original model h is based solely on W , in the new model we add another matrix called D , representing the vectors of paragraphs:



A paragraph token can be thought of as yet another word token except that (at least in the paper) all paragraph vectors are unique, whereas word tokens share their vector representations among different contexts. At each step we compute h by concatenating or averaging a paragraph vector d with a context of word vectors C :

$$h = \frac{1}{C} \cdot (D_d + W_1 + W_W + \dots + W_C)$$

The weight update functions are the same as in Word2Vec except that we now also update the paragraph vectors. This first model is called the Distributed Memory Model of Paragraph Vectors. Le & Mikolov present another model called Distributed Bag of Words Model of Paragraph Vector. This model ignores the context C and attempts to predict a randomly sampled word from a randomly sampled context window.

Let's have a more detailed look at the DM Model. In addition to the matrix W we need to randomly initialize a matrix D with dimensions $P \times N$ where P is the number of paragraphs or whatever textual unit we use, and N is the number of dimensions.

In [27]:

```
V, N, P = len(vocabulary), 3, 5
WI = (np.random.random((V, N)) - 0.5) / N
WO = (np.random.random((N, V)) - 0.5) / V
D = (np.random.random((P, N)) - 0.5) / N
```

Say our corpus consists of the following five sentences (paragraphs):

In [28]:

```
sentences = ['snowboarding is dangerous', 'skydiving is dangerous',
             'escargots are tasty to some people', 'everyone loves tasty food',
             'the minister has some dangerous ideas']
```

We first convert the sentences into a vectorial BOW representation:

In [29]:

```
vocabulary = Vocabulary()
sentences_bow = list(docs2bow(sentences, vocabulary))
sentences_bow
```

Out[29]:

```
[[0, 1, 2],
 [3, 1, 2],
 [4, 5, 6, 7, 8, 9],
 [10, 11, 6, 12],
 [13, 14, 15, 8, 2, 16]]
```

Next we compute the posterior probability for each word in the vocabulary given the concatenation and averaging of the first paragraph and the context word *snowboarding*. We compute the error and update the hidden-to-output layer weights.

In [30]:

```
target_word = 'dangerous'
h = (D[0] + WI[vocabulary['snowboarding']]) / 2
learning_rate = 1.0

for word in vocabulary:
    p = (np.exp(np.dot(WO.T[vocabulary[word]], h)) /
          sum(np.exp(np.dot(WO.T[vocabulary[w]], h)) for w in vocabulary))
    t = 1 if word == target_word else 0
    error = t - p
    WO.T[vocabulary[word]] = (WO.T[vocabulary[word]] - learning_rate * error *
                               vocabulary[word])
print WO
```

```
[[-0.01599941  0.02135301  0.0927715  -0.00565041 -0.00361651 -0.
01454073
  0.02333261  0.00211833  0.00254255  0.02315315 -0.01917578  0.
00724787
 -0.00117272 -0.02043504  0.00593186 -0.0166333  -0.0306218 ]
 [ 0.0089237  -0.00397806 -0.13199195  0.02555059 -0.02095756 -0.
00978333
  0.01561624  0.03603476 -0.02114407 -0.01552016  0.01289922  0.
00119743
 -0.00112818  0.01708133  0.00765248  0.02442374  0.01109005]
 [-0.01205008 -0.03123478  0.05878695  0.02615259 -0.01025209 -0.
00442044
  0.00311309  0.01554668  0.02344194  0.00602561 -0.03117694  0.
01368817
  0.00858936 -0.00223242 -0.01141366 -0.01719967 -0.01400046]]
```

We backpropagate the error to the input-to-hidden layer as follows:

In [31]:

```
EH = WO.sum(1)
WI[vocabulary['snowboarding']] = WI[vocabulary['snowboarding']] - 0.5 * learning_rate * EH
D[0] = D[0] - 0.5 * learning_rate * EH
```

Experiments

Le & Mikolov evaluate and investigate the performance of the paragraph vectors on a number of different tasks. I will briefly discuss them here.

Sentiment Analysis

In the first experiment, the authors address the task of sentiment analysis. They make use of the Stanford Sentiment Treebank Dataset which is a manually annotated data set containing 11855 sentences taken from the movie review site Rotten Tomatoes. Each sentence in the data set has been assigned a label on a scale of negative to positive. The task is to predict these labels.

Le & Mikolov train Paragraph Vectors using both the DM Model and the DBOW model. These two representations are concatenated for each training instance and fed to a Logistic Regression classifier that makes a prediction for each unseen test sentence. They compare the performance of the model to a number of different models:

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%

Le & Mikolov then move on beyond the sentence level and evaluate their model on the IMDB data set. Training and testings follows the same procedure as before. The results presented below suggest a strong improvement compared to the other reported models:

Model	Error rate
BoW (bnc) (Maas et al., 2011)	12.20 %
BoW (b Δ t'c) (Maas et al., 2011)	11.77%
LDA (Maas et al., 2011)	32.58%
Full+BoW (Maas et al., 2011)	11.67%
Full+Unlabeled+BoW (Maas et al., 2011)	11.11%
WRRBM (Dahl et al., 2012)	12.58%
WRRBM + BoW (bnc) (Dahl et al., 2012)	10.77%
MNB-uni (Wang & Manning, 2012)	16.45%
MNB-bi (Wang & Manning, 2012)	13.41%
SVM-uni (Wang & Manning, 2012)	13.05%
SVM-bi (Wang & Manning, 2012)	10.84%
NBSVM-uni (Wang & Manning, 2012)	11.71%
NBSVM-bi (Wang & Manning, 2012)	8.78%
Paragraph Vector	7.42%

Information Retrieval

The authors then turn to an experiment in Information Retrieval. They develop a task in which the goal is to predict which of three paragraphs isn't the result of the same query. They construct a data set on the basis of the search results of a search engine. For each query they create a triplet of paragraphs: two paragraphs are results from the same query and one is randomly sampled from the rest of the collection (result from a different query). The pairwise distances between each member of a triplet, should then reflect which two results belong to the same query and which snippet is the outlier. The following table shows quite convincingly how well the paragraph vectors are able to perform on this task compared to the other methods:

Model	Error rate
Vector Averaging	10.25%
Bag-of-words	8.10 %
Bag-of-bigrams	7.28 %
Weighted Bag-of-bigrams	5.67%
Paragraph Vector	3.82%


Comments / Points of critique

General remarks

Although the proposed model is only a small step beyond the original word2vec model (and some of the writings is quite sloppy), I think it is a really clever one with much potential for many different applications. The ease with which the model can be applied to text pieces of variable length is perhaps the strongest advantage of the model.

Availability of code / experimentation details

No implementation was available (even not to the second author, which led to some serious doubts about the reproducibility of the results:

 **Tomas Mikolov** 23-09-14

★ **Andere ontvangers:** chris...@cs.brown.edu, zbu...@uci.edu

[Bericht vertalen in het Nederlands](#)

I tried myself to reproduce Quoc's results during the summer; I could get error rates on the IMDB dataset to around 9.4% - 10% (depending on how good the text normalization was). However, I could not get anywhere close to what Quoc reported in the paper (7.4% error, that's a huge difference).

This was part of one intern's project. Unfortunately, he did not finish his work yet, so although we have permission to publish the code I wrote + paper about additional stuff, there has been no progress on this in the last two months. I was originally planning to extend word2vec code to support the sentence vectors, but until I will be able to reproduce the results, I am not going to change the main word2vec version. Of course we also asked Quoc about the code; he promised to publish it but so far nothing has happened. I am starting to think that Quoc's results are actually not reproducible.

Tomas
- tekst uit het oorspronkelijke bericht weergegeven -

After a nice series of discussion, the first author finally made a suggestion on how to improve the results and actually come to the same performance as reported in the paper:

★ Adriaan Schakel	@Tomas: Thanks for sharing your code! In this connection, I would like to bring to the attention of this community another	06-10-14
★ Hubert Soyer	Thank you very much, this is much better than what I was hoping for! I'll have a thorough look at the code. Thanks again! Best, Hubert	07-10-14
★ adam...@gmail.com	Hello, I'm the author of the zseymour implementation. I've not touched the repo in some time due to comp. exams, but I certainly	30-10-14
★ oklian...@gmail.com	Hi, I'm using the doc2vec to reproduce the experiments on the Stanford Sentiment Tree bank dataset of this paper. I trained the	10-11-14
★ Tomas Mikolov	I don't know what doc2vec is, but you can use my code that I posted here in this thread on Oct 6th to get to ~90% accuracy on the IMDB	11-11-14
★ oklian...@gmail.com	Hi, Tomas! Thanks for your quick reply! The code runs well except the Liblinear should be a newer version (i.e., 1.95). Yes, for the	11-11-14
★ oklian...@gmail.com	Sorry for the incorrect result in my previous email. I can get 73% accuracy for the Stanford Sentiment Tree Bank dataset with this	15-11-14
★ q...@google.com		23-12-14
★	Andere ontvangers: chris...@cs.brown.edu, zbu...@uci.edu, hubert...@gmail.com	

[Bericht vertalen in het Nederlands](#)

A simple modification to the word2vec training command should give better results. Try changing line 55 of your go.sh to this command:

```
time ./word2vec -train ../alldata-id.txt -output vectors.txt -cbow 0 -size 100 -window 10 -negative 5 -hs 1 -sample 1e-3 -threads 40 -binary 0 -iter 20 -min-count 1 -sentence-vectors 1
```

On my machine, I got 92.6% on the dataset. Combining with rnnlm features should get better than 93%. If you combine PV-DM and PV-DBOW, you should get similar results as well.

On Monday, October 6, 2014 9:07:32 AM UTC-7, Tomas Mikolov wrote:

```
> I'm sending modified word2vec version that I wrote during the summer to help one intern with his project. It allows to train the sentence vectors, and the attached script runs it on IMDB. It also trains recurrent neural network language model to perform classification (another baseline, showing that generative models can work reasonably well for this task too, although the discriminative ones are obviously better). You can comment out this part of the script.
>
>
```

It is applaudable for the authors to interact and respond to comments by the readers. On the other hand, if they would have developed an implementation similar to the original word2vec, this discussion (and the resulting doubts in the method) wouldn't have to take place. The authors report a number of hyperparameters in the paper, such as the window size and the number of dimensions of the vectors. However, some crucial parameters weren't mentioned (such as the use of hierarchical softmax versus negative sampling). Again, in order to reproduce the results, the authors must specify in much greater detail how they performed the experiments.

In []:

In [1]:

```
from IPython.core.display import HTML
def css_styling():
    styles = open("custom.css", "r").read()
    return HTML(styles)
css_styling()
```

Out[1]:

```
/* Placeholder for custom user CSS mainly to be overridden in
profile/static/custom/custom.css This will always be an empty file in IPython */
```

In []: