Word2Vec and FastText Word Embedding with Gensim

黃功詳 Steeve Huang
Feb 4, 2018 · 7 min read

(https://iwcollege-cdn-pull-zone-theisleofwightco1.netdna-ssl.com/wp-content/uploads/2017/05/2DGerAvyRM.jpg)

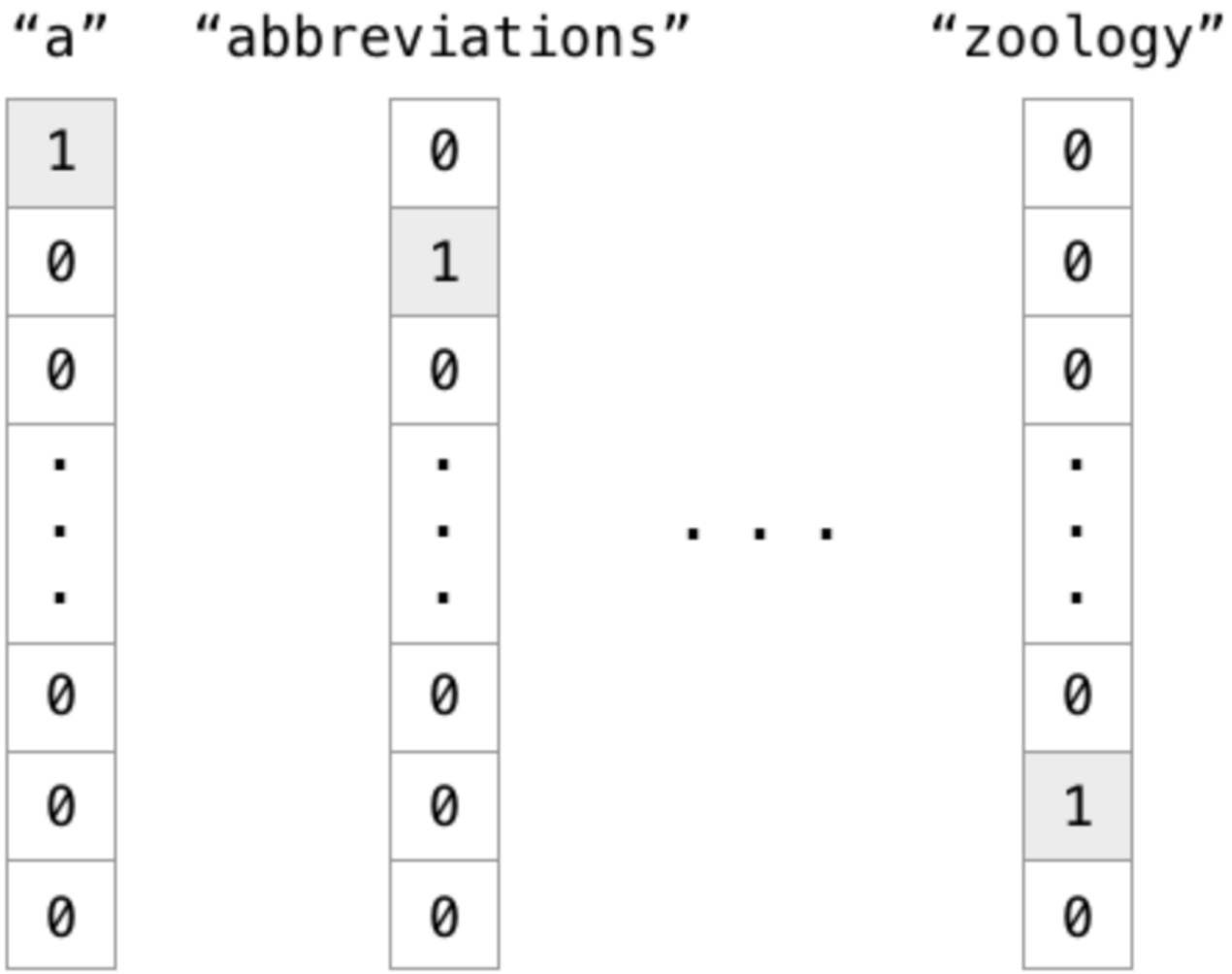
In Natural Language Processing (NLP), we often map words into vectors that contains numeric values so that machine can understand it. Word embedding is a type of mapping that allows words with similar meaning to have similar representation. This article will introduce two state-of-the-art word embedding methods, **Word2Vec** and **FastText** with their implementation in Gensim.

. . .

Traditional Approach

A traditional way of representing words is one-hot vector, which is essentially a vector with only one target element being 1 and the others being 0. The length of the vector is

equal to the size of the total unique vocabulary in the corpora. Conventionally, these unique words are encoded in alphabetical order. Namely, you should expect the one-hot vectors for words starting with “a” with target “1” of lower index, while those for words beginning with “z” with target “1” of higher index.



https://cdn-images-1.medium.com/max/1600/1*ULfyiWPKgWceCqyZeDTl0g.png

Though this representation of words is simple and easy to implement, there are several issues. First, you cannot infer any relationship between two words given their one-hot representation. For instance, the word “endure” and “tolerate”, although have similar meaning, their targets “1” are far from each other. In addition, sparsity is another issue as there are numerous redundant “0” in the vectors. This means that we are wasting a lot of space.We need a better representation of words to solve these issues.

Word2Vec

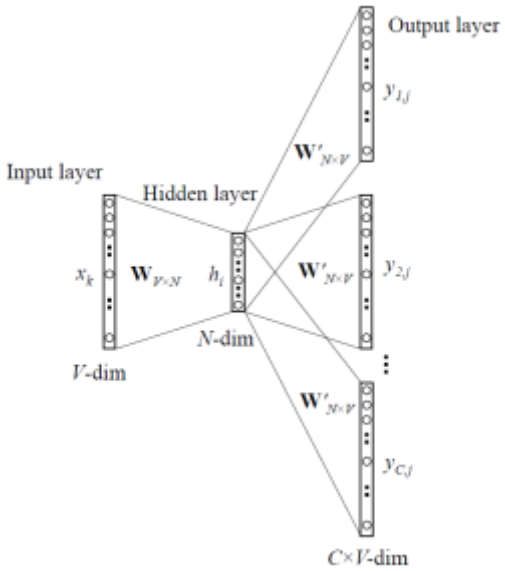
Word2Vec is an efficient solution to these problems, which leverages the context of the target words. Essentially, we want to use the surrounding words to represent the target

words with a Neural Network whose hidden layer encodes the word representation.

There are two types of Word2Vec, Skip-gram and Continuous Bag of Words (CBOW). I will briefly describe how these two methods work in the following paragraphs.

Skip-gram

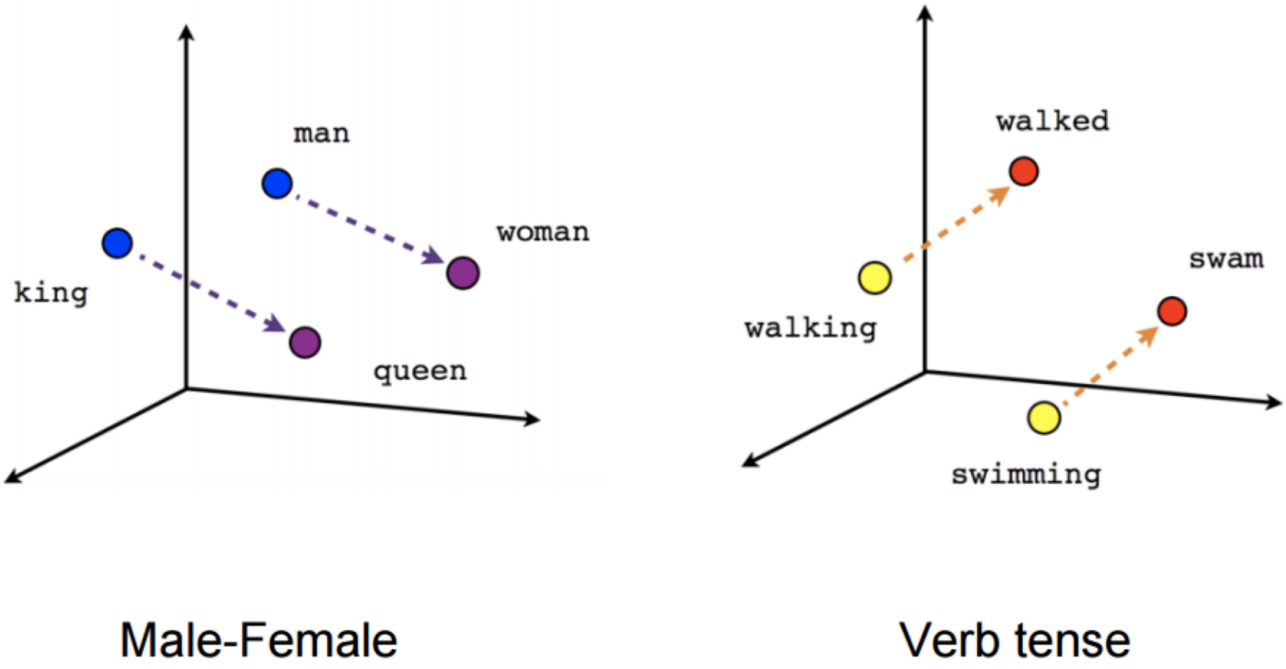
For skip-gram, the input is the target word, while the outputs are the words surrounding the target words. For instance, in the sentence "*I have a cute dog*", the input would be "*a*", whereas the output is "*I*", "*have*", "*cute*", and "*dog*", assuming the window size is 5. All the input and output data are of the same dimension and one-hot encoded. The network contains 1 hidden layer whose dimension is equal to the embedding size, which is smaller than the input/ output vector size. At the end of the output layer, a softmax activation function is applied so that each element of the output vector describes how likely a specific word will appear in the context. The graph below visualizes the network structure.



Skip-gram (https://www.analyticsvidhya.com/blog/2017/06/word-embeddings-count-word2veec/)

The word embedding for the target words can obtained by extracting the hidden layers after feeding the one-hot representation of that word into the network.

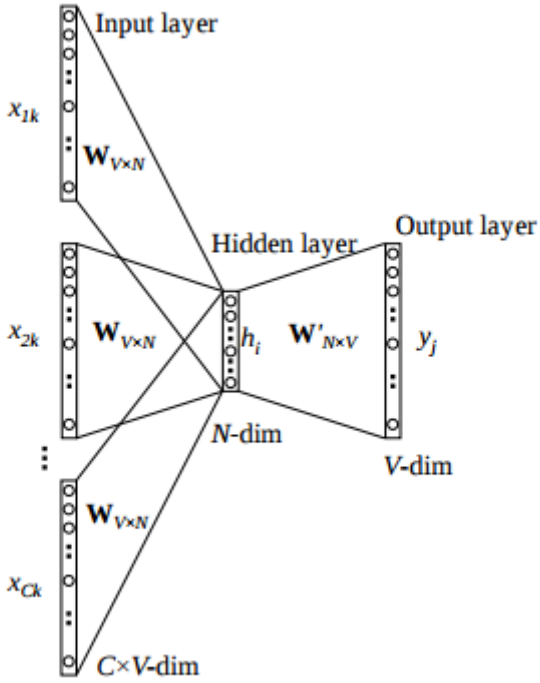
With skip-gram, the representation dimension decreases from the vocabulary size (V) to the length of the hidden layer (N). Furthermore, the vectors are more "meaningful" in terms of describing the relationship between words. The vectors obtained by subtracting two related words sometimes express a meaningful concept such as gender or verb tense, as shown in the following figure (dimensionality reduced).



Visualize Word Vectors (https://www.tensorflow.org/images/linear-relationships.png)

CBOW

Continuous Bag of Words (CBOW) is very similar to skip-gram, except that it swaps the input and output. The idea is that given a context, we want to know which word is most likely to appear in it.



CBOW (https://www.analyticsvidhya.com/blog/2017/06/word-embeddings-count-word2veec/)

The biggest difference between Skip-gram and CBOW is that **the way the word vectors are generated**. For CBOW, all the examples with the target word as target are fed into the networks, and taking the average of the extracted hidden layer. For example, assume we only have two sentences, “He is a nice guy” and “She is a wise queen”. To compute the word representation for the word “a”, we need to feed in these two examples, “He is nice guy”, and “She is wise queen” into the Neural Network and take the average of the value in the hidden layer. Skip-gram only feed in the one and only one target word one-hot vector as input.

It is claimed that Skip-gram tends to do better in rare words. Nevertheless, the performance of Skip-gram and CBOW are generally similar.

Implementation

I will show you how to perform word embedding with Gensim, a powerful NLP toolkit, and a TED Talk dataset.

First, we download the the dataset using urllib, extracting the subtitle from the file.

```
import numpy as np
import os
from random import shuffle
import re
import urllib.request
import zipfile
import lxml.etree
#download the data
urllib.request.urlretrieve("https://wit3.fbk.eu/get.php?path=XML_releases/xml/ted_en-2(
# extract subtitle
with zipfile.ZipFile('ted_en-20160408.zip', 'r') as z:
    doc = lxml.etree.parse(z.open('ted_en-20160408.xml', 'r'))
input_text = '\n'.join(doc.xpath('//content/text()'))
```

gistfile1.txt hosted with ❤ by **GitHub** **view raw**

Let’s take a look at what *input_text* variable stores, as partially shown in the following figure.

input_text

Clearly, there are some redundant information that is not helpful for us to understand the talk, such as the words describing sound in the parenthesis and the speaker's name. We get rid of these words with regular expression.

```
# remove parenthesis
input_text_noparens = re.sub(r'\([^)]*\)', '', input_text)
# store as list of sentences
sentences_strings_ted = []
for line in input_text_noparens.split('\n'):
    m = re.match(r'^(?:(?P<precolon>[^:]{,20}):)?(?P<postcolon>.*)$', line)
    sentences_strings_ted.extend(sent for sent in m.groupdict()['postcolon'].split('.')
# store as list of lists of words
sentences_ted = []
for sent_str in sentences_strings_ted:
    tokens = re.sub(r"[^a-z0-9]+", " ", sent_str.lower()).split()
    sentences_ted.append(tokens)
```

gistfile1.txt hosted with ❤ by **GitHub** **view raw**

Now, *sentences_ted* has been transformed into a two dimensional array with each element being a word. Let's print out the first and the second element.

sentences_ted

This is the form that is ready to be fed into the Word2Vec model defined in Gensim. Word2Vec model can be easily trained with one line as the code below.

```
from gensim.models import Word2Vec
model_ted = Word2Vec(sentences=sentences_ted, size=100, window=5, min_count=5, workers=4
```

gistfile1.txt hosted with ❤ by **GitHub** **view raw**

- *sentences*: the list of split sentences.
- *size*: the dimensionality of the embedding vector
- *window*: the number of context words you are looking at
- *min_count*: tells the model to ignore words with total count less than this number.

- *workers*: the number of threads being used
- *sg*: whether to use skip-gram or CBOW

Now, let's try which words are most similar to the word "man".

```
model_ted.wv.most_similar("man")
```


It appears words related to men/women/kid are most similar to "man".

Although Word2Vec successfully handles the issue posed by one-hot vector, it has several limitation. The biggest challenge is that it is not able to represent words that do not appear in the training dataset. Even though using a larger training set that contains more vocabulary, some rare words used very seldom can never be mapped to vectors.

FastText

FastText is an extension to Word2Vec proposed by Facebook in 2016. Instead of feeding individual words into the Neural Network, FastText breaks words into several n-grams (sub-words). For instance, the tri-grams for the word *apple* is *app, ppl,* and *ple* (ignoring the starting and ending of boundaries of words). The word embedding vector for *apple* will be the sum of all these n-grams. After training the Neural Network, we will have word embeddings for all the n-grams given the training dataset. Rare words

can now be properly represented since it is highly likely that some of their n-grams also appears in other words. I will show you how to use FastText with Gensim in the following section.

Implementation

Similar to Word2Vec, we only need one line to specify the model that trains the word embedding.

```
1  from gensim.models import FastText
2  model_ted = FastText(sentences_ted, size=100, window=5, min_count=5, workers=4,sg=1)
```


Let's try it with the word *Gastroenteritis*, which is rarely used and does not appear in the training dataset.

```
1  model_ted.wv.most_similar("Gastroenteritis")
```


Even though the word *Gastroenteritis* does not exist in the training dataset, it is still capable of figuring out this word is closely related to some medical terms. If we try this in the Word2Vec defined previously, it would pop out error because such word does not exist in the training dataset. Although it takes longer time to train a FastText model

(number of n-grams > number of words), it performs better than Word2Vec and allows rare words to be represented appropriately.

Conclusion

You have learned what Word2Vec and FastText are as well as their implementation with Gensim toolkit. Should you have any problem, feel free to leave a comment below. If you like this article, make sure you follow me on twitter so you won't miss out any great Machine Learning/ Deep Learning blog post!

Machine Learning NLP Word Embeddings Word2vec Towards Data Science