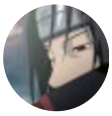
Stepping into NLP — Word2Vec with Gensim

Introduction to word2vec embeddings and use cases



Vijay athithya Follow

Feb 27 · 5 min read

Photo by Dmitry Ratushny on Unsplash

Natural language processing (NLP) is an area of computer science and [artificial intelligence](https://hackernoon.com/) that is *k*nown to be concerned with the interaction between computer and hum*a*ns in n*a*tural language. The goal is to enable the systems to fully understand *v*arious language as well as we do. It is the driving force behind NLP products/techniques like virtual assistants, speech recognition, machine translation, sentiment analysis, automatic text summarization, and much more. We'll be working on *a* word embedding technique called Word2Vec using Gensim framework in this post.

Word Embeddings… what!!

Word Embedding is an NLP technique, capable of capturing the context of a word in a document, semantic and syntactic similarity, relation with other words, etc. In general, they are vector representations of a particular word. Having said that what follows is the techniques to create Word Embeddings. There are many techniques to create Word Embeddings. Some of the popular ones are:

1. Binary Encoding.
2. TF Encoding.
3. TF-IDF Encoding.
4. Latent Semantic Analysis Encoding.
5. Word2Vec Embedding.

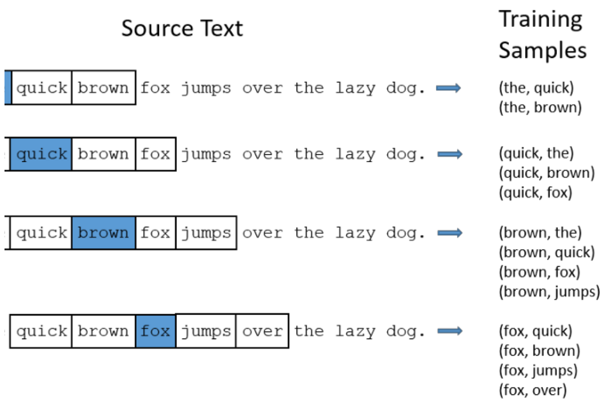
We'll discuss the different embedding techniques on future posts, for now, we'll stick with Word2Vec Embedding.

Intro to Word2Vec Embedding

Word2vec is one of the most widely used models to produce word embeddings. These models are shallow, two-layer neural networks that are trained to reconstruct linguistic contexts of words. Word2Vec can be implemented in two ways, one is Skip Gram and other is Common Bag Of Words (CBOW)

Continuous Bag Of Words (CBOW)

CBOW is learning to predict the word by the context. Here the input will be the context ***#neighboring words*** and output will be the target word. The limit on the number of words in each context is determined by a parameter called "**window size**".



Example: The quick brown fox jumps over the lazy dog ***#yes the same example :-)***

Model: CBOW

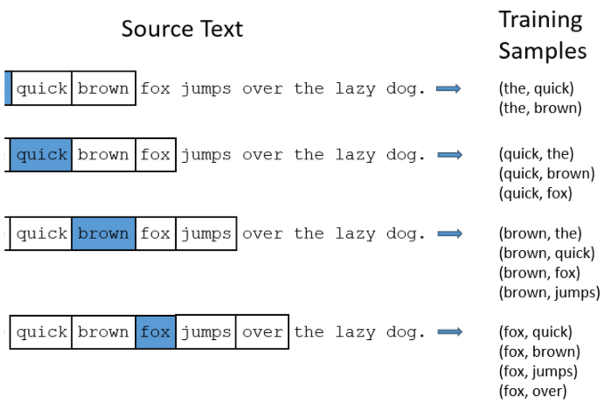
INPUT Layer: White box content

TARGET Layer: blue box word

Window Size: 5

Skip Gram

Skip Gram is learning to predict the context by the word. Here the input will be the word and output will be the target context ***#neighboring words***. The limit on the number of words in each context is determined by a parameter called "**window size**".



Example: The quick brown fox jumps over the lazy dog ***#yes the same example :-)***

Model: Skip Gram

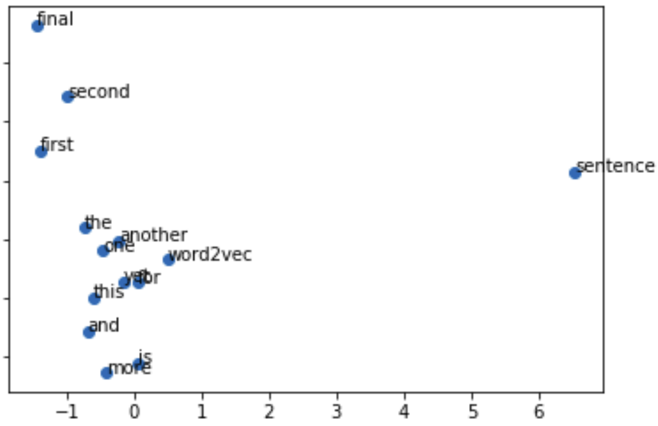
INPUT Layer: blue box word

TARGET Layer: White box content
Window Size: 5

Behind the scenes

- As discussed above, we'll be using two-layer neural networks. For this model, the input later will be the context respect to target word followed by the hidden layer which constructs the relationship lastly the target layer with the target word.
- After training the model, each word in the corpus will have its own vector embeddings with respect to the context and meaning.
- Now we can use *Matplotlib* for mapping the word embeddings, which might give us a clear picture of how relationships are made

and vectors are assigned.



Word2Vec Embeddings

```
INPUT CORPUS

1. this is the first
sentence for word2vec

2. this is the second
sentence

3. yet another sentence

4. one more sentence

5. and the final sentence
```

- As we can see similar words are mapped nearby based on their context like 'first' & 'second', 'one' & 'another' and the word 'sentence' is separated from the clusters as it is nowhere similar to any of the other words.
- From here we can use these embeddings to have similar words, sentence or documents with the same content and the list goes on…

Literally everywhere!!

That's it for explaining things, I believe ***you*** have got some understanding about word2vec. If not, don't worry! you can get a clear idea after going through the example below. let's dive into some python 🐍.

Let's add Some Python

As we discussed earlier, we'll be implementing word2vec using Gensim framework in python. **Gensim** is a robust open-source vector space modeling and topic modeling toolkit implemented in Python. It uses NumPy, SciPy and optionally Cython for performance.

```
# Word2vec model for embeddings
from gensim.models import Word2Vec
# For extracting pre-trained vectors
from gensim.models import KeyedVectors
# PCA for dimensionality reduction
from sklearn.decomposition import PCA
# For ploting the results
```

Here we have imported the requirements, next we'll be defining out text corpus.

```
# define training data
sentences = [['this', 'is', 'the', 'first', 'sentence',
             ['this', 'is', 'the', 'second', 'sentence']
             ['yet', 'another', 'sentence'],
             ['one', 'more', 'sentence'],
```

In this step, We are defining our **Word2vec model**.

```
# Defining the structure of our word2vec model

# Size is the dimentionality feature of the model
model_1 = Word2Vec(size=300, min_count=1)
#Feeding Our coupus
model_1.build_vocab(sentences)
#Lenth of the courpus
```

Here,

- `size` is the dimensionality of the vector higher the size denser the embeddings(ideally `size` must be lower than the vocab length. Using a higher dimensionality than vocabulary size would more-or-less guarantee 'overfitting'.)
- Words below the `min_count` frequency are dropped before training occurs(as we have few lines of input corpus, we are taking every word).
- Corpus is added to vocab for training and training is done.

PCA model is used to reduce the **n** dimensioned vector for each word in our vocab to a 2d vector(we are doing this for plotting/visualize our results).

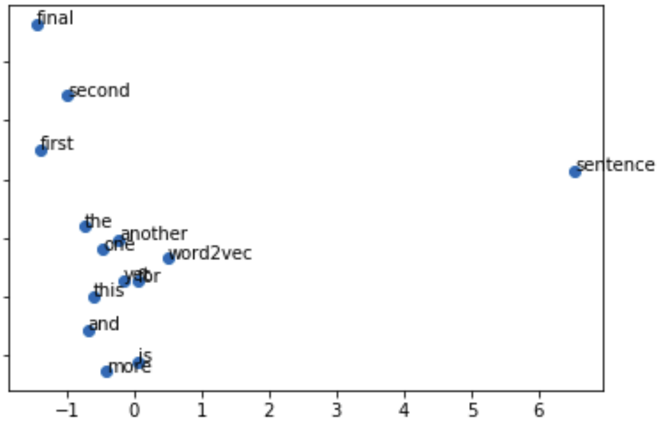
```
# fit a 2d PCA model to the vectors

# X holds the vectors of n dimentions for each word in
X = model_1[model_1.wv.vocab]

# We are reducing the n dimentions to 2d
pca = PCA(n_components=2)
result = pca.fit_transform(X)
```

pca.py hosted with ❤ by GitHub view raw

We'll be using scatter plot in matplotlib for plotting the word embeddings,

```python
# create a scatter plot of the projection
pyplot.scatter(result[:, 0], result[:, 1])
words = list(model_1.wv.vocab)
for i, word in enumerate(words):
    pyplot.annotate(word, xy=(result[i, 0], result[i, 1
pyplot.show()
```

visualize.py hosted with ❤ by GitHub view raw



As we can see, the word embeddings are mapped relative to each other. This result is had, by using the pre-trained model for simplicity basic model is discussed. Original model can be found here.

Let's find some similarity

Using this word embeddings we can find similarity between the words in our corpus.

```
>>>model_1.most_similar(positive=['first'], topn=1)

[('second', 0.8512464761734009)]
```

The most_similar function is to find similar words in our embeddings to the target word.

```
>>>model_1.similarity('one', 'another')

0.80782
```

Here we found the similarity between the two words in our embeddings. Like this, there are many useful functions to work with. They can be found here.

. . .

We end this here, hope I've given some introduction to the word2vec embeddings. Check the other works here.

Lol, if you think so we are on the same page. Let's connect Medium, Linkedin, Facebook.