# **Lab 5**

# **Word Embeddings in NLP | Word2Vec | GloVe | fastText**

* **Word embeddings are word vector representations where words with similar meaning have similar representation.**
* Word vectors are one of the most efficient ways to represent words.

# **Word Vectors**

* Word vectors are much better ways to represent words than **one hot encoded vector** (As the size of vocabulary increases, leads to extensive memory usage while representing text vectors).
* In one hot encoded vectors, the vectors for “dog” and “cat” are just as close to each other as “dog” and “computer”, hence neural network has to try really hard to understand each word since they are being treated as completely isolated entities.
* The usage of word vectors aim to resolve both these issues.

# **Word2Vec**

* In word2vec there are 2 architectures **CBOW** (Continuous Bag of Words) and **Skip Gram**.
* First thing to do is to collect word co-occurrence data.
* Consider, “Deep Learning is very hard and fun”. We need to set something known as **window size**. Let’s say 2 in this case.
* Here since our window size is 2 we will consider 2 words behind the word and 2 words after the word, hence each word will get 4 words associated with it.
* As we are **passing the context window** through the text data, we find all **pairs of target and context** words to form a dataset in the format of target word and context word. For the sentence above, it will look like this:

**1st Window pairs:** (Deep, Learning), (Deep, is)

**2nd Window pairs:** (Learning, Deep), (Learning, is), (Learning, very)

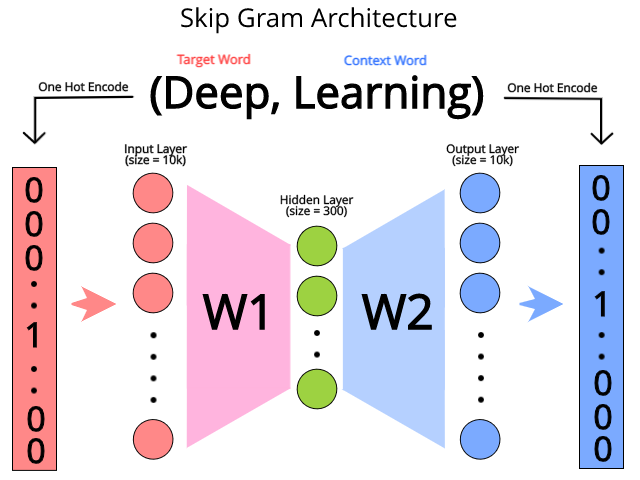
**3rd Window pairs:** (is, Deep), (is, Learning), (is, very), (is, hard)

Deep, Learning), (Deep, is), (Learning, Deep), (Learning, is), (Learning, very), (is, Deep), (is, Learning), (is, very), (is, hard), (very, learning), (very, is), (very, hard), (very, and), (hard, is), (hard, very), (hard, and), (hard, fun), (and, very), (and, hard), (and, fun), (fun, hard), (fun, and)

This can be considered as our **“training data”** for word2vec.

**Skip gram model:**

* In skip gram model, we try to predict each context word given a target word.
* We use neural network for this prediction task.
* **The input to the neural network is the one hot encoded version of the context word**.
* Hence the size of the input and output layer is **V**(vocabulary count).
* This neural network has only one layer in the middle, the **size of the hidden layer determines the size of the word vectors we wish to have at the end.**



* **Neural network here is trying to guess which context words can appear given a target word.**
* After training the neural network, if we input any target word into the neural network, it will give a vector output which represents the words which have a high probability of appearing near the given word.
* **For CBOW the only difference is that we try to predict the target word given the context words**, essentially we just invert the skip gram model to get the CBOW model

**Word vectors help represent semantics of the words**

* **Word2Vec** takes as its input a large corpus of text and produces a vector space, typically of several hundred dimensions, with each unique word in the corpus being assigned a corresponding vector in the space.
* Word vectors are positioned to vector space such that words that share common contexts in the corpus are located in close proximity to one other in the space

## **Implementation**

**word embedding with Gensim**

|  |
| --- |
| 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-20160408.zip&filename=ted\_en-20160408.zip", filename="ted\_en-20160408.zip") |
|  |

|  |
| --- |
| # 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()'))

|  |
| --- |
| # 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('.') if sent) |
|  |

|  |
| --- |
| # 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)

* sentences\_ted has been transformed into a two dimensional array with each element being a word.
* Word2Vec model can be easily trained with one line as the code below.

|  |
| --- |
| from gensim.models import Word2Vec |
|  |

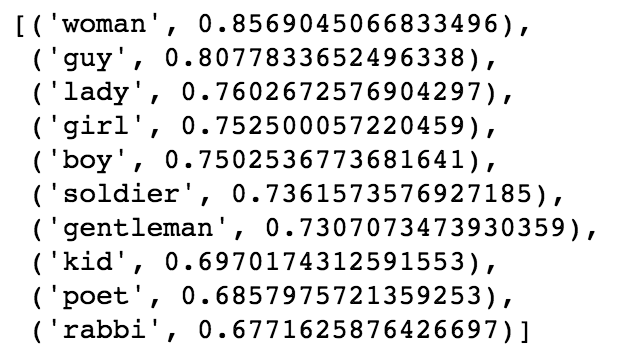
model\_ted = Word2Vec(sentences=sentences\_ted, size=100, window=5, min\_count=5, workers=4, sg=0)

* *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”)

output



**2.FastText**

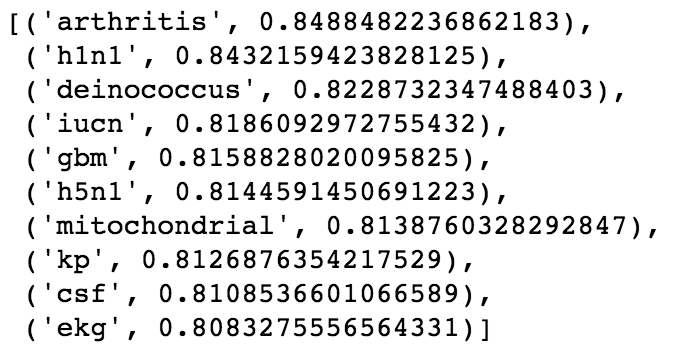
* FastText is an extension to Word2Vec proposed by Facebook in 2016.
* Instead of feeding individual words into the Neural Network, FastText breaks words into several n-grams (sub-words).
* For instance, the tri-grams for the word apple is app, ppl, and ple (ignoring the starting and ending of boundaries 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

|  |
| --- |
| from gensim.models import FastText |
|  |

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.

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.

# **3.Glove**

* Glove is based on **matrix factorization technique on word context matrix**.
* It first **constructs** a **large matrix** of (words x context) co-occurrence information ie. for each word, you count how frequently we see those word in some context in a large corpus.
* In order to understand how GloVe works, we need to understand 2 main methods which GloVe was built on

**Global matrix factorization**

* In NLP, global matrix factorization is the process of using matrix factorization form linear algebra to reduce large term frequency matrices.
* These matrices usually represent the occurrences or the absence of words in the document.

**Local context window.**

* Local context window methods are **CBOW** and **Skip-Gram**,
* **Glove is a word vector representation method where training is performed on aggregated global word-word co-occurrence statistics from the corpus**.
* This means that like word2vec it uses context to understand and create the word representations.

**Intuition:**

* Instead of learning the **raw occurrence probabilities,** it was more useful to learn **ratios of these co-occurrence probabilities**.
* The embeddings are optimized, so that **the dot product of 2 vectors equals the log of number of times the 2 words will occur near each other.**
* For example, if 2 words “cat” and “dog” occur in the context of each other, say 20 times in 10-word window in the document corpus, then —
* **Vector (cat).Vector (dog) = log (10)**
* This forces the model to encode the frequency distribution of words that occur near them in a more global context.

Implementation

Step 1: Install Libraries

|  |
| --- |
| pip install glove\_python |
|  |

|  |
| --- |
| import nltk |
|  |

|  |
| --- |
| nltk.download('stopwords') |
|  |

|  |
| --- |
| nltk.download ('punkt') |
|  |

nltk.download('wordnet')

Step 2: Define the Input Sentence

lines= ["Hello this is a tutorial to convert word to integer" , "It is a beautiful day" , "Jack is going to office"]

Step 3: Tokenize

|  |
| --- |
| from nltk.tokenize import sent\_tokenize, word\_tokenize |
|  |

|  |
| --- |
| word\_tokens=[] |
|  |

|  |
| --- |
| i=0 |
|  |

|  |
| --- |
| for line in lines: |
|  |

|  |
| --- |
| words = word\_tokenize(line) |
|  |

|  |
| --- |
| word\_tokens.insert(i,words) |
|  |

|  |
| --- |
| i=i+1 |
|  |

print (word\_tokens)

Step 4: Stop Word Removal

|  |
| --- |
| from nltk.corpus import stopwords |
|  |

|  |
| --- |
| stop\_words=stopwords.words('english') |
|  |

|  |
| --- |
| lines\_without\_stopwords=[] |
|  |

|  |
| --- |
| for line in lines: |
|  |

|  |
| --- |
| stop\_removed=[] |
|  |

|  |
| --- |
| for line in word\_tokens: |
|  |

|  |
| --- |
| for word in line: |
|  |

|  |
| --- |
| if word not in stop\_words: |
|  |

|  |
| --- |
| stop\_removed.append(word) |
|  |

print (stop\_removed)

Step 5: Lemmatize

|  |
| --- |
| from nltk import WordNetLemmatizer |
|  |

|  |
| --- |
| from nltk.stem import WordNetLemmatizer |
|  |

|  |
| --- |
| wordnet\_lemmatizer = WordNetLemmatizer() |
|  |

|  |
| --- |
| lines\_with\_lemmas=[] #stop words contain the set of stop words |
|  |

|  |
| --- |
| for line in lines: |
|  |

|  |
| --- |
| lem\_line=[] |
|  |

|  |
| --- |
| for word in stop\_removed: |
|  |

|  |
| --- |
| lem\_line.append(wordnet\_lemmatizer.lemmatize(word)) |
|  |

|  |
| --- |
| string='' |
|  |

|  |
| --- |
| new\_lines=','.join([str(i) for i in lem\_line]) |
|  |

|  |
| --- |
| print (lem\_line) |
|  |

print (new\_lines)

**Step 6: Building model**

|  |
| --- |
| #importing the glove library |
|  |

|  |
| --- |
| from glove import Corpus, Glove |
|  |

|  |
| --- |
| # creating a corpus object |
|  |

|  |
| --- |
| corpus = Corpus() |
|  |

|  |
| --- |
| #training the corpus to generate the co occurence matrix which is used in GloVe |
|  |

|  |
| --- |
| corpus.fit(new\_lines, window=10) |
|  |

|  |
| --- |
| #creating a Glove object which will use the matrix created in the above lines to create embeddings |
|  |

|  |
| --- |
| #We can set the learning rate as it uses Gradient Descent and number of components |
|  |

|  |
| --- |
| glove = Glove(no\_components=5, learning\_rate=0.05) |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
| glove.fit(corpus.matrix, epochs=30, no\_threads=4, verbose=True) |
|  |

|  |
| --- |
| glove.add\_dictionary(corpus.dictionary) |
|  |

glove.save('glove.model')

**Step 7: Evaluate the model**

print glove.word\_vectors[glove.dictionary['samsung']]