National University of Computer and Emerging Sciences, Islamabad.

# Natural Language Processing (NLP)

## Assignment 4&5

**Word2Vec Doc2Vec and FastText**

| | |
|---|---|
| NAME | REGISTRATION NUMBER |
| Nasir Iqbal | 17I-0519 |
| DEGREE PROGRAM | BS(CS) |
| Section | B |
| SUBMITTED TO: | **Sir, Muhammad Bin Arif** |

## Table of Contents

## Abstract

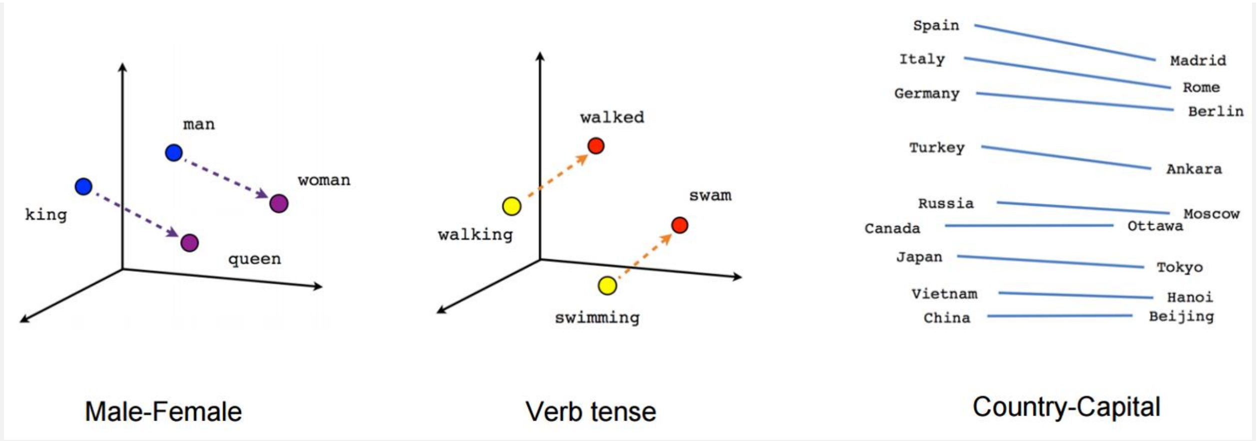
This report has four parts. In the first part i discuss word embeddings. I discuss the need for them, some of the methods to create them, and some of their interesting properties. In the second part I discuss about word2vec. Furthermore, I have explained about how to implement it from scratch using Pytorch. Third part of the report comprise of how Doc2vec works and how it is different from or similar to word2vec.In the last part I have discuss about FastText. How does it works. I also explained about how does it solves the problem of unseen words.

# Word Embedding

## What is Word Embedding?

Word Embeddings are a numerical vector representation of the text in the corpus that maps each word in the corpus vocabulary to a set of real valued vectors in a pre-defined N-dimensional space.

Deep learning techniques such as neural network models are used to learn these vector-representations for each word in the corpus vocabulary. Based on the usage of these word in sentences, the Word Embeddings attempt to catch the semantic, contextual, and syntactic meaning of each word in the corpus vocabulary. Words that have similar semantic and contextual meaning also have similar vector representations while at the same time each word in the vocabulary will have a unique set of vector representation.



Source: https://www.tensorflow.org/tutorials/representation/word2vec

The graphic above shows how to map words in a 3-Dimensional Vector Space that have identical contextual, semantic, and grammatical meanings. We can see that the vector differences between the word pairs (walking & walked) and (swimming & swam) are about similar in the above image example of Verb Tense.

## Why Word Embedding?

In almost all NLP tasks input is given in the form of text data. Managing text information is risky, since our PCs, scripts and AI models can't peruse and comprehend text in any human sense.

Many various associations come to mind**(Ayush, 2020)**. when I hear the word "cat": it's a small fluffy animal that's cute, eats fish, isn't allowed by my landlord, and so on. These language linkages, on the other hand, are the result of extremely complex neural calculations developed through millions of years of evolution, whereas our machine learning models must start from scratch with no prior knowledge of word meaning.

So, how should textual input be sent to our models? Computers can handle numerical input well, so let’s rephrase the question to:

***How can we best numerically represent textual input?***

Whatever numerical representation approach we come up with should ideally be semantically meaningful, capturing as much of the linguistic meaning of a word as possible. A well-chosen, informative input representation can make a significant difference in overall model performance.

**Word embeddings** are the dominant approach to this problem, and are so pervasive that their use is practically assumed in any NLP project **(Karani, 2018)**. Whether you're working on text categorization, sentiment analysis, or machine translation, you'll almost certainly begin by downloading pre-calculated embeddings (if your problem is relatively standard) or thinking about which method to use to calculate your own word embeddings from your dataset.
In this report I will show how to implement Word2Vec which is one of the  popular embedding technique.

## Word2Vec

### What is Word2Vec?

In 2013, Tomas Mikolov, et al. at Google developed the Word2Vec algorithm. The distributional hypothesis was used to create the algorithm. According to the **distributional hypothesis**, words that appear in similar linguistic situations will have comparable semantic meanings **(Lakhey, 2019).** This idea is used by Word2Vec to map words with comparable semantic meanings that are geometrically close to each other in a N-Dimensional vector space. Word2Vec rebuilds the linguistic context of words by training a series of shallow, 2-layer neural networks **(Bednerski, 2018).** It takes a big corpus of text as an input and outputs a vector space with hundreds of dimensions. Each distinct word in the corpus vocabulary is allocated a distinct vector in the space.

It can be implemented using either of the two techniques: **Common Bag of Words(CBOW)** or **Skip Gram**.

But in this report we will cover skip gram only.

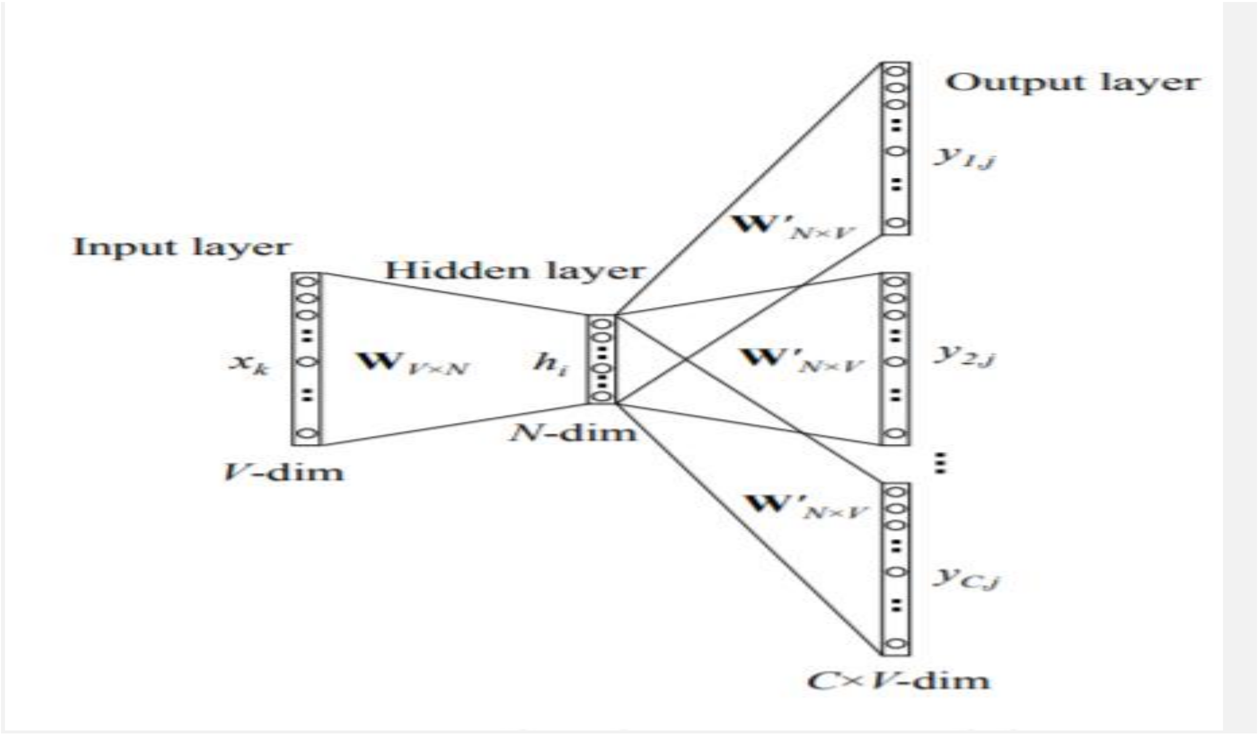
#### Skip Gram

We send a single word to the model in skip grams, and the model tries to predict the words that surround it. The output is a series of probability distributions for each word in the vocabulary, with the input being a one-hot-encoded vector of the word. for example. "I am going for a walk,"

```
The vocabulary is — [“I”, “am”, “going”, “for”, “a”, “walk”].
Length of Vocabulary V=6
```

For setting the number of surrounding words that model tries to predict, we define Context Window.
Let Context Window C=4

Let the input **(Ayush, 2020)** be one of the vocabulary words. We'll provide the model the one-hot-encoded representation of this word in dimension V, and it should return a series of probability distributions for each word in the output dimension C*V.



Source: word2vec Parameter Learning Explained by Xin Rong

Word2Vec was a big development in the field of Word Embeddings since it was able to capture previously uncaptured relationships in algebraic representations. For example, if we mapped words like "King," "Queen," "man," and "wife" into vector space, we discovered that the vector distance between "King" and "Queen" was the same as the vector distance between "man" and "wife", which could allow us to produce outputs like the following:

Word2Vec["King"] - Word2Vec["man"] + Word2Vec["woman"] = Word2Vec["Queen"]

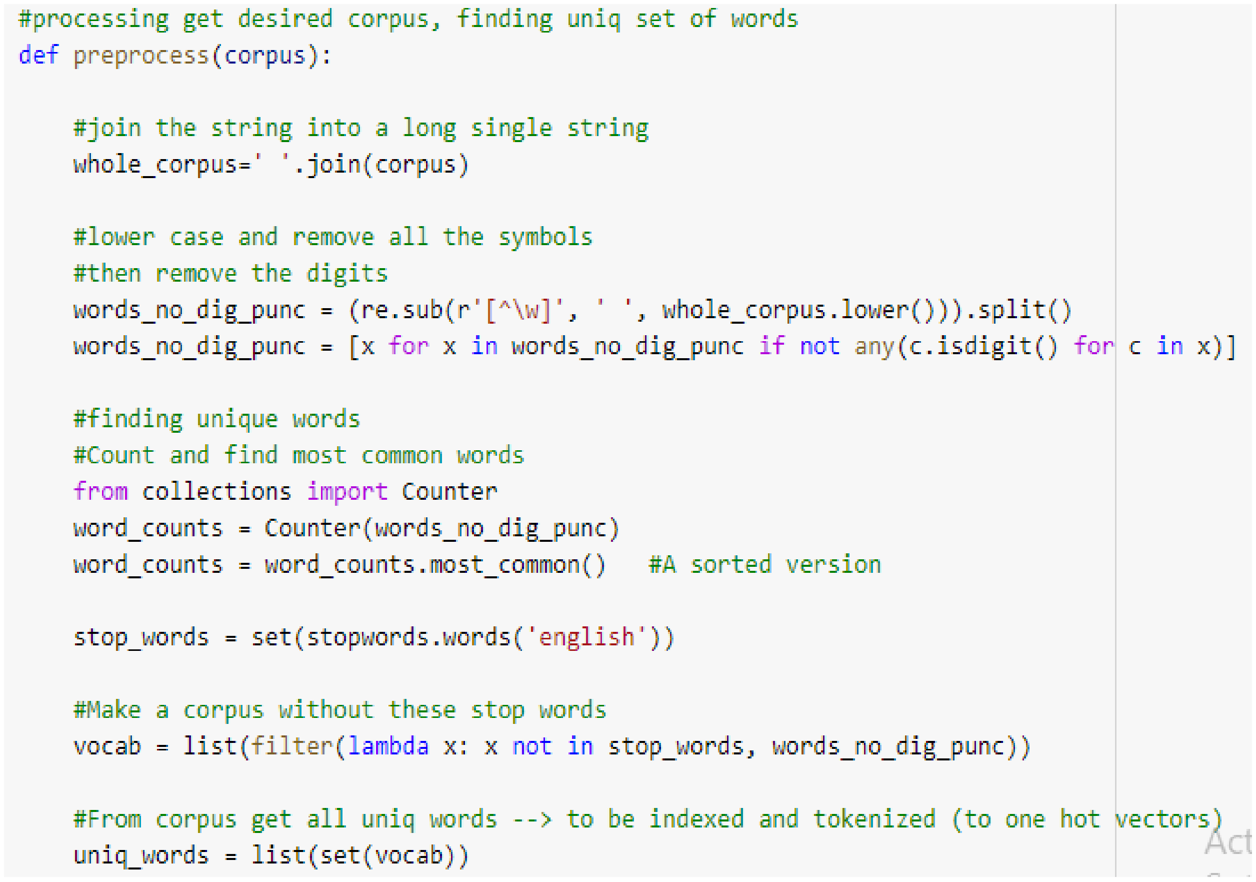
## Word2Vec Implementation using Skip Gram with Negative Sampling

**Data preprocessing and Creating vocabulary**

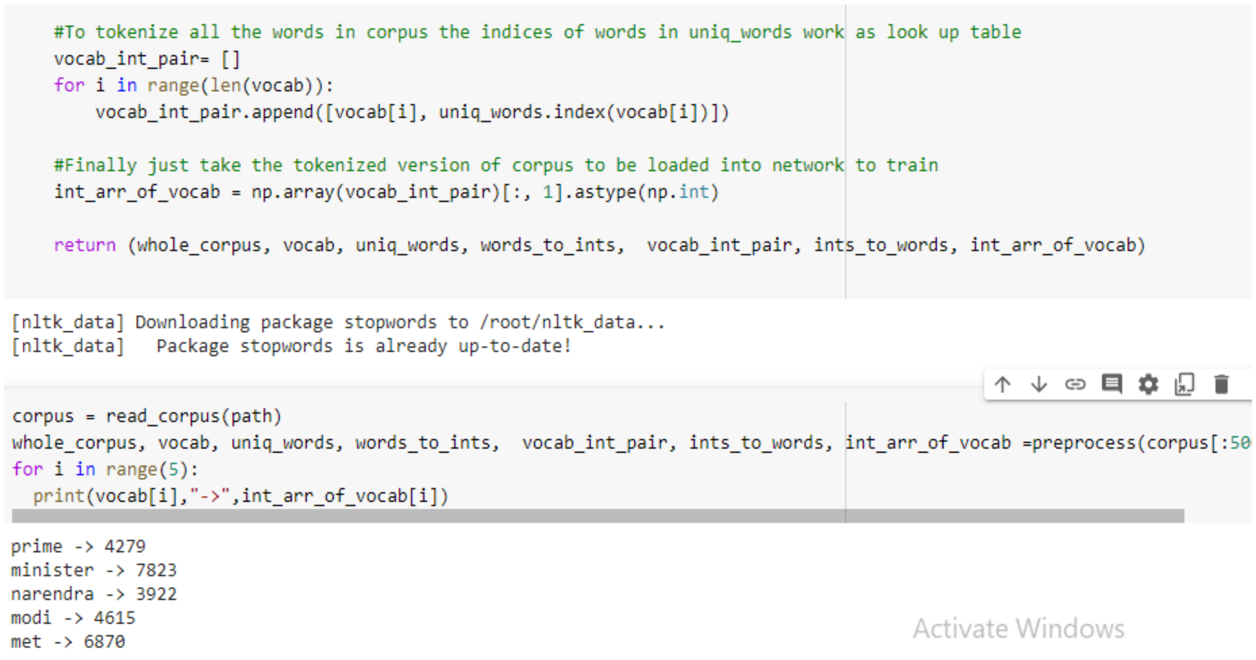
The very first step in any NLP task is to preprocess the data. Because many times it happens that the input data is not cleaned and it effect the accuracy of model. In Preprocessing step first, we will tokenize the corpus using regular expression library of python. Next step is to convert each word into lower case. Till now we have a list of lower case tokenize words. Now remove the stop words from the list.

And Finally, create the vocabulary from that list. Vocabulary is basically a list of unique words with assigned indices.

The Corpus I am using is a speech of Indian prime minister “Narender Modi”. The corpus consist of 11000 sentences but we will be using first 400 sentences for the model training.



```python
#processing get desired corpus, finding uniq set of words
def preprocess(corpus):

    #join the string into a long single string
    whole_corpus=' '.join(corpus)

    #lower case and remove all the symbols
    #then remove the digits
    words_no_dig_punc = (re.sub(r'[^\w]', ' ', whole_corpus.lower())).split()
    words_no_dig_punc = [x for x in words_no_dig_punc if not any(c.isdigit() for c in x)]

    #finding unique words
    #Count and find most common words
    from collections import Counter
    word_counts = Counter(words_no_dig_punc)
    word_counts = word_counts.most_common()   #A sorted version

    stop_words = set(stopwords.words('english'))

    #Make a corpus without these stop words
    vocab = list(filter(lambda x: x not in stop_words, words_no_dig_punc))

    #From corpus get all uniq words --> to be indexed and tokenized (to one hot vectors)
    uniq_words = list(set(vocab))
```



```
    #To tokenize all the words in corpus the indices of words in uniq_words work as look up table
    vocab_int_pair= []
    for i in range(len(vocab)):
        vocab_int_pair.append([vocab[i], uniq_words.index(vocab[i])])

    #Finally just take the tokenized version of corpus to be loaded into network to train
    int_arr_of_vocab = np.array(vocab_int_pair)[:, 1].astype(np.int)

    return (whole_corpus, vocab, uniq_words, words_to_ints,  vocab_int_pair, ints_to_words, int_arr_of_vocab)
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

```
corpus = read_corpus(path)
whole_corpus, vocab, uniq_words, words_to_ints,  vocab_int_pair, ints_to_words, int_arr_of_vocab =preprocess(corpus[:50
for i in range(5):
  print(vocab[i],"->",int_arr_of_vocab[i])
```

```
prime -> 4279
minister -> 7823
narendra -> 3922
modi -> 4615
met -> 6870
```

Activate Windows

Here I have print the first 5 words and their corresponding indexes in the vocablory.

**Negative Sampling**

A neural network is trained by taking a training sample and slightly modifying all of the neuron weights so that it more accurately predicts that training sample. In other words, each training sample will change all of the neural network's weights.

Because of the vastness of our word lexicon, our skip-gram neural network has a large number of weights, all of which would be somewhat changed by every one of our billions of training samples!

Negative sampling addresses this by having each training sample only modify a small percentage of the weights, rather than all of them. Here's how it works **(Lakhey, 2018)**.

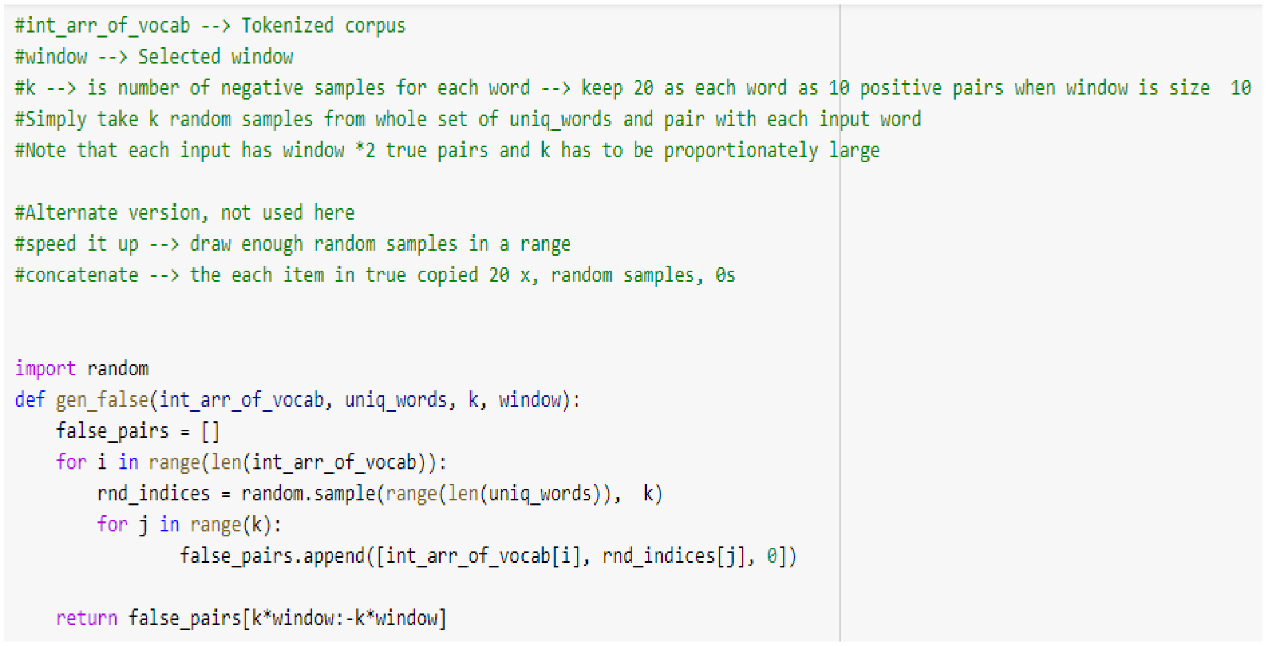
Remember that the network's "label" or "correct output" is a one-hot vector when training it on the word pair ("fox", "quick"). That is, the output neuron corresponding to "quick" should output a 1 while the rest of the hundreds of output neurons should produce a 0.

Instead, we'll use negative sampling to select a small number of "negative" words (let's say 5) at random to update the weights for. (A "negative" word is one for which we want the network to output a 0 in this context.) We'll also continue to update the weights for our "positive" term (in this case, the word "fast").The research paper on word2vec says that selecting 5–20 words works well for smaller datasets, and you can get away with only 2–5 words for large datasets.

Notice that our model's output layer has a weight matrix of 300 x 10,000? So all we'll do is update the weights for our positive word ("fast"), as well as the weights for five other words we want to yield 0. There are 6 output neurons in all, with 1,800 weight values. That's merely 0.06 percent of the output layer's 3M weights!

In the hidden layer, only the weights for the input word are updated (this is true whether you're using Negative Sampling or not).

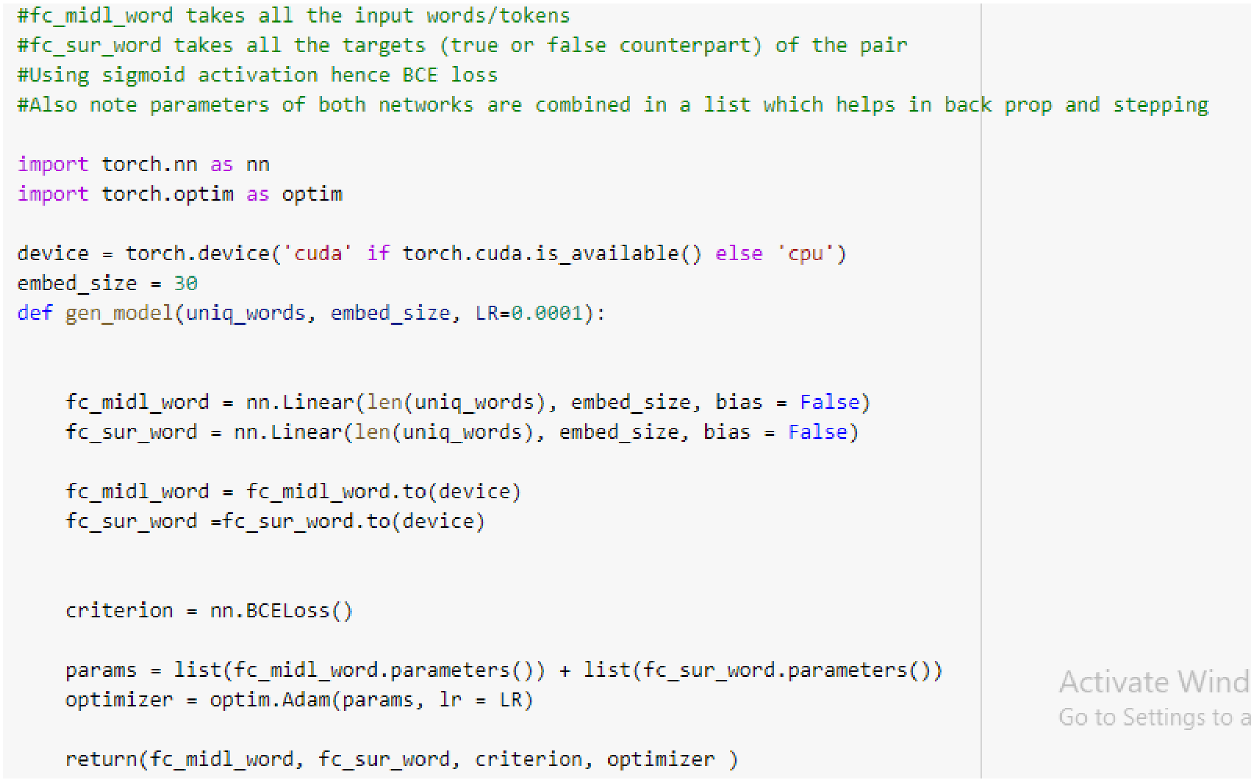
Now lets implement the negative sampling:



```python
#int_arr_of_vocab --> Tokenized corpus
#window --> Selected window
#k --> is number of negative samples for each word --> keep 20 as each word as 10 positive pairs when window is size  10
#Simply take k random samples from whole set of uniq_words and pair with each input word
#Note that each input has window *2 true pairs and k has to be proportionately large

#Alternate version, not used here
#speed it up --> draw enough random samples in a range
#concatenate --> the each item in true copied 20 x, random samples, 0s


import random
def gen_false(int_arr_of_vocab, uniq_words, k, window):
    false_pairs = []
    for i in range(len(int_arr_of_vocab)):
        rnd_indices = random.sample(range(len(uniq_words)),  k)
        for j in range(k):
                false_pairs.append([int_arr_of_vocab[i], rnd_indices[j], 0])

    return false_pairs[k*window:-k*window]
```

## Network

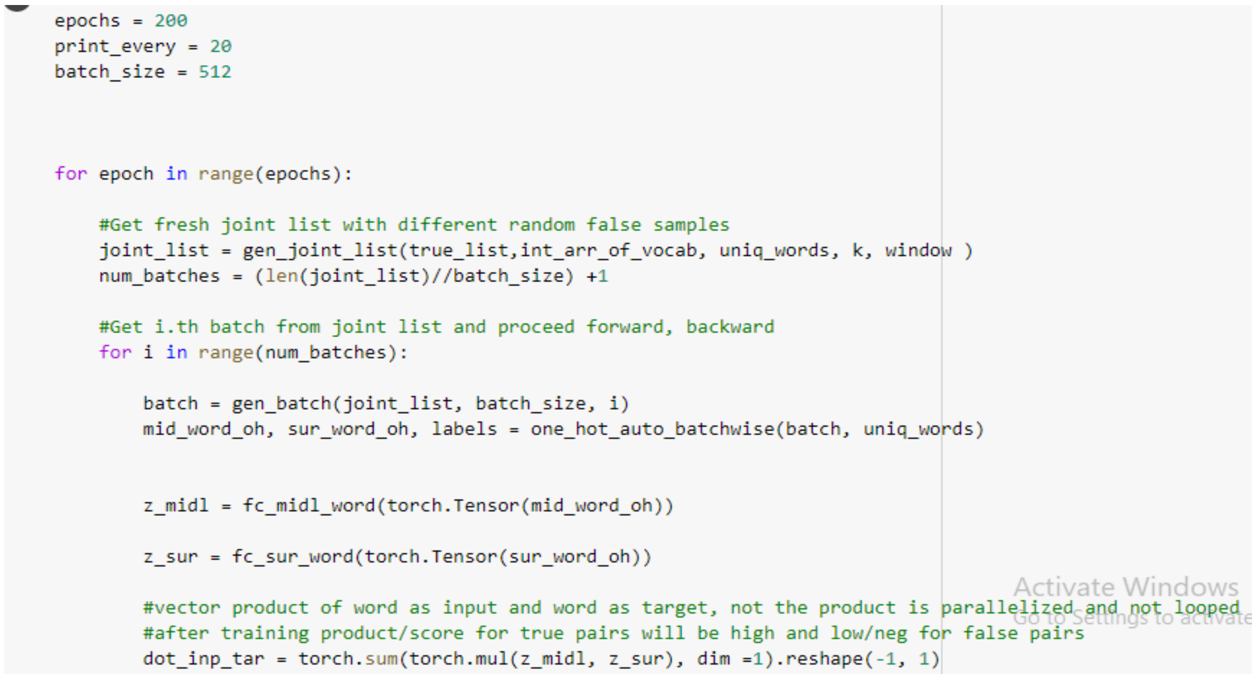
Finally, we will define the network for our model. Our model has two fully connected linear layers.I have combined the parameters of both the networks in a list so that it help in back propagation and stepping.



```
#fc_midl_word takes all the input words/tokens
#fc_sur_word takes all the targets (true or false counterpart) of the pair
#Using sigmoid activation hence BCE loss
#Also note parameters of both networks are combined in a list which helps in back prop and stepping

import torch.nn as nn
import torch.optim as optim

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
embed_size = 30
def gen_model(uniq_words, embed_size, LR=0.0001):


    fc_midl_word = nn.Linear(len(uniq_words), embed_size, bias = False)
    fc_sur_word = nn.Linear(len(uniq_words), embed_size, bias = False)

    fc_midl_word = fc_midl_word.to(device)
    fc_sur_word =fc_sur_word.to(device)


    criterion = nn.BCELoss()

    params = list(fc_midl_word.parameters()) + list(fc_sur_word.parameters())
    optimizer = optim.Adam(params, lr = LR)

    return(fc_midl_word, fc_sur_word, criterion, optimizer )
```

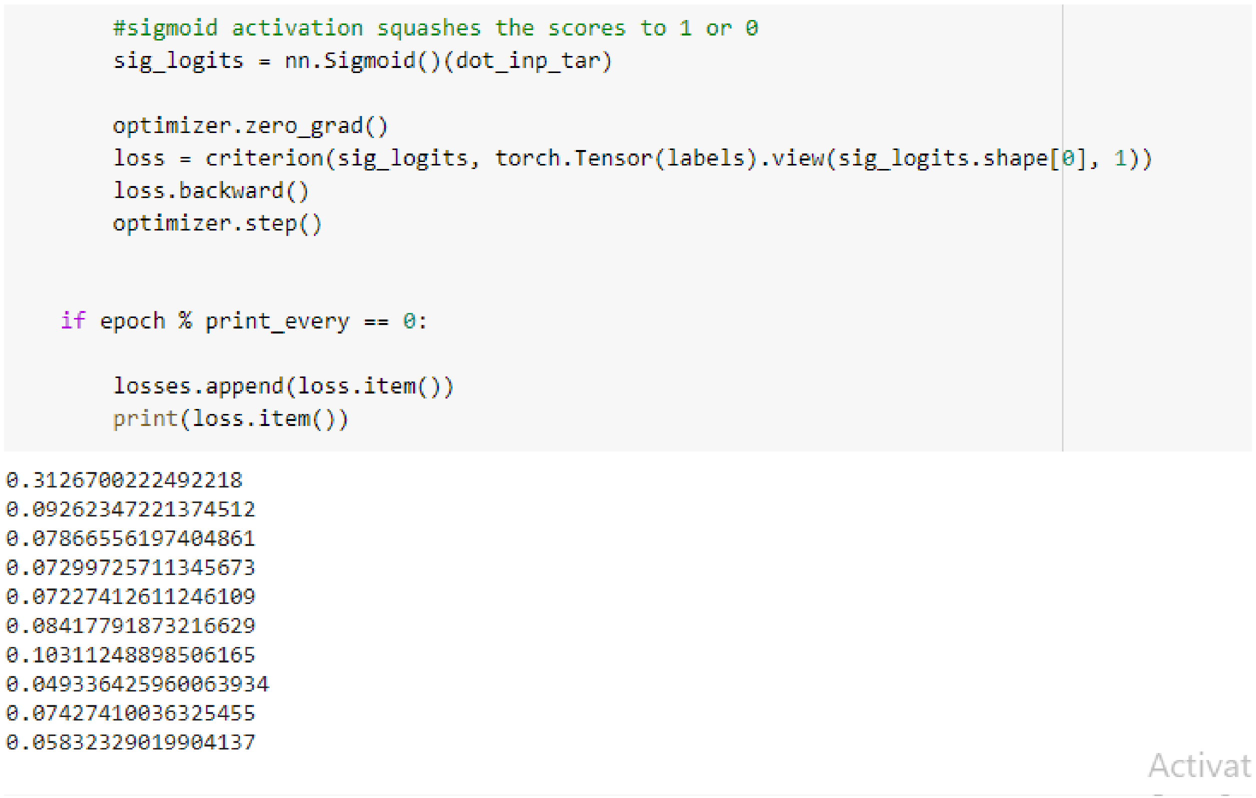
Here for generating model I have used the linear layers of nn module in pytorch.

## Model Training



```
epochs = 200
print_every = 20
batch_size = 512



for epoch in range(epochs):

    #Get fresh joint list with different random false samples
    joint_list = gen_joint_list(true_list,int_arr_of_vocab, uniq_words, k, window )
    num_batches = (len(joint_list)//batch_size) +1

    #Get i.th batch from joint list and proceed forward, backward
    for i in range(num_batches):

        batch = gen_batch(joint_list, batch_size, i)
        mid_word_oh, sur_word_oh, labels = one_hot_auto_batchwise(batch, uniq_words)


        z_midl = fc_midl_word(torch.Tensor(mid_word_oh))

        z_sur = fc_sur_word(torch.Tensor(sur_word_oh))

        #vector product of word as input and word as target, not the product is parallelized and not looped
        #after training product/score for true pairs will be high and low/neg for false pairs
        dot_inp_tar = torch.sum(torch.mul(z_midl, z_sur), dim =1).reshape(-1, 1)
```

Activate Windows
Go to Settings to activate

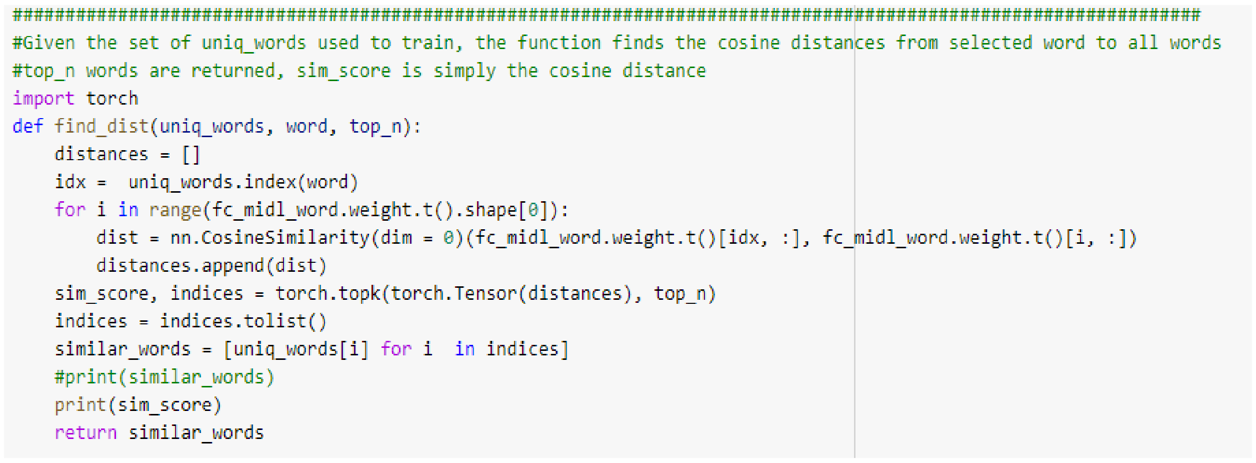


```
        #sigmoid activation squashes the scores to 1 or 0
        sig_logits = nn.Sigmoid()(dot_inp_tar)

        optimizer.zero_grad()
        loss = criterion(sig_logits, torch.Tensor(labels).view(sig_logits.shape[0], 1))
        loss.backward()
        optimizer.step()


    if epoch % print_every == 0:

        losses.append(loss.item())
        print(loss.item())
```

```
0.3126700222492218
0.09262347221374512
0.07866556197404861
0.07299725711345673
0.07227412611246109
0.08417791873216629
0.10311248898506165
0.049336425960063934
0.07427410036325455
0.05832329019904137
```

Activat

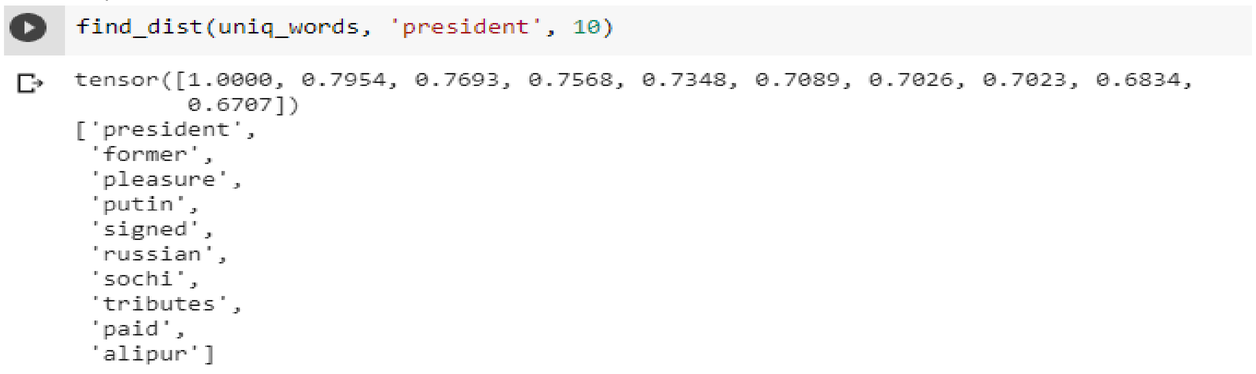
## Cosine Similarity



```
##################################################################################################################
#Given the set of uniq_words used to train, the function finds the cosine distances from selected word to all words
#top_n words are returned, sim_score is simply the cosine distance
import torch
def find_dist(uniq_words, word, top_n):
    distances = []
    idx =  uniq_words.index(word)
    for i in range(fc_midl_word.weight.t().shape[0]):
        dist = nn.CosineSimilarity(dim = 0)(fc_midl_word.weight.t()[idx, :], fc_midl_word.weight.t()[i, :])
        distances.append(dist)
    sim_score, indices = torch.topk(torch.Tensor(distances), top_n)
    indices = indices.tolist()
    similar_words = [uniq_words[i] for i  in indices]
    #print(similar_words)
    print(sim_score)
    return similar_words
```

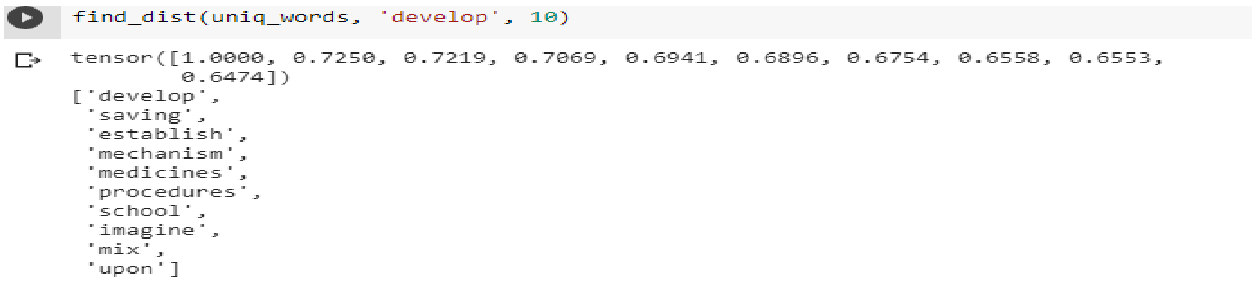
## Output Comparison with Different window sizes

First output with window size 5 is attached below:



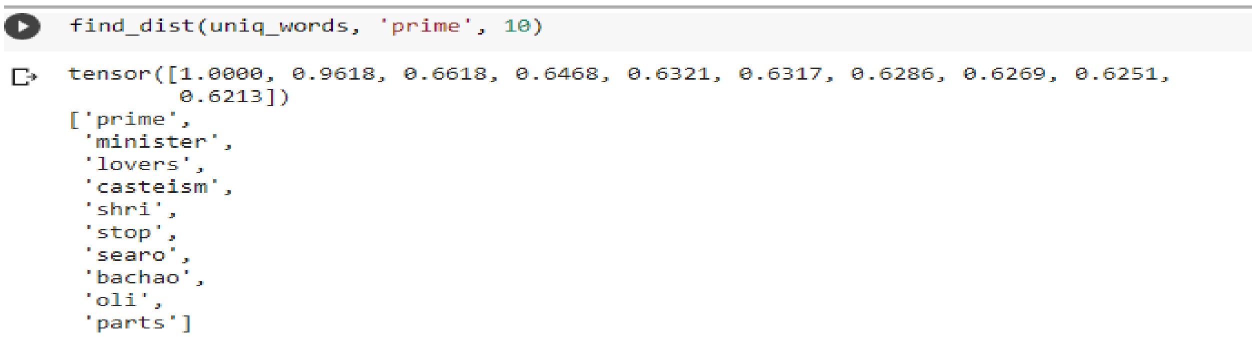
```
find_dist(uniq_words, 'president', 10)
```

```
tensor([1.0000, 0.7954, 0.7693, 0.7568, 0.7348, 0.7089, 0.7026, 0.7023, 0.6834,
        0.6707])
['president',
 'former',
 'pleasure',
 'putin',
 'signed',
 'russian',
 'sochi',
 'tributes',
 'paid',
 'alipur']
```



```
find_dist(uniq_words, 'develop', 10)
```

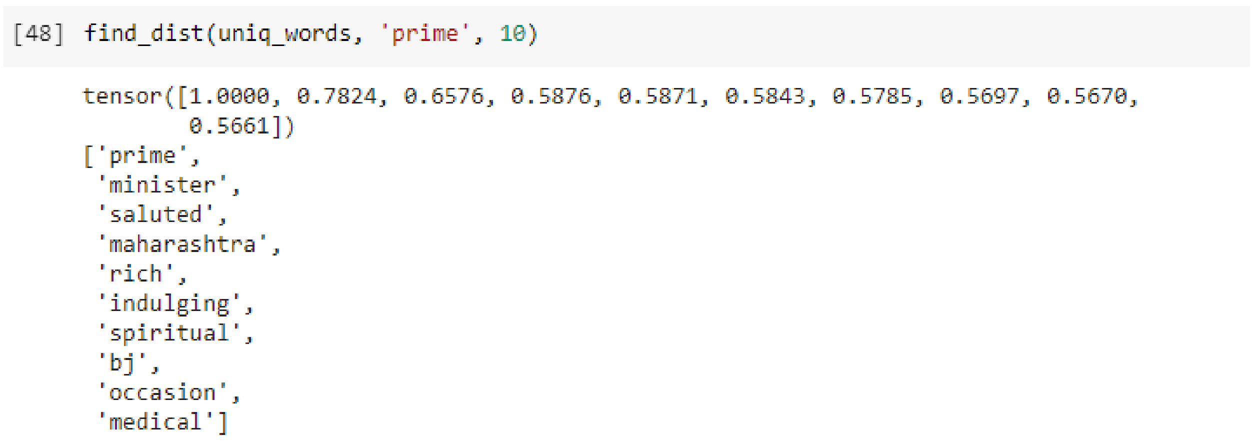
```
tensor([1.0000, 0.7250, 0.7219, 0.7069, 0.6941, 0.6896, 0.6754, 0.6558, 0.6553,
        0.6474])
['develop',
 'saving',
 'establish',
 'mechanism',
 'medicines',
 'procedures',
 'school',
 'imagine',
 'mix',
 'upon']
```



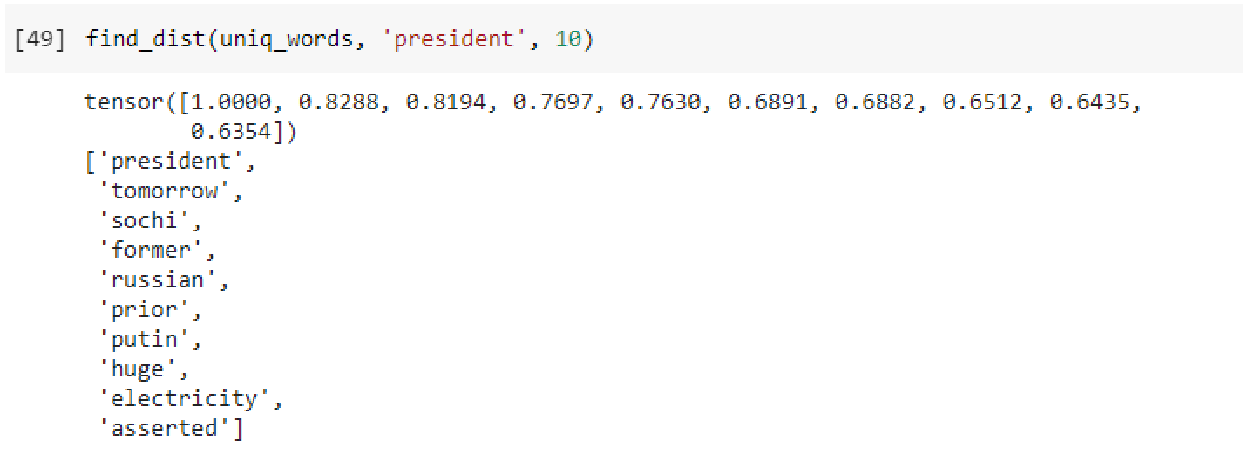
```
find_dist(uniq_words, 'prime', 10)
```

```
tensor([1.0000, 0.9618, 0.6618, 0.6468, 0.6321, 0.6317, 0.6286, 0.6269, 0.6251,
        0.6213])
['prime',
 'minister',
 'lovers',
 'casteism',
 'shri',
 'stop',
 'searo',
 'bachao',
 'oli',
 'parts']
```

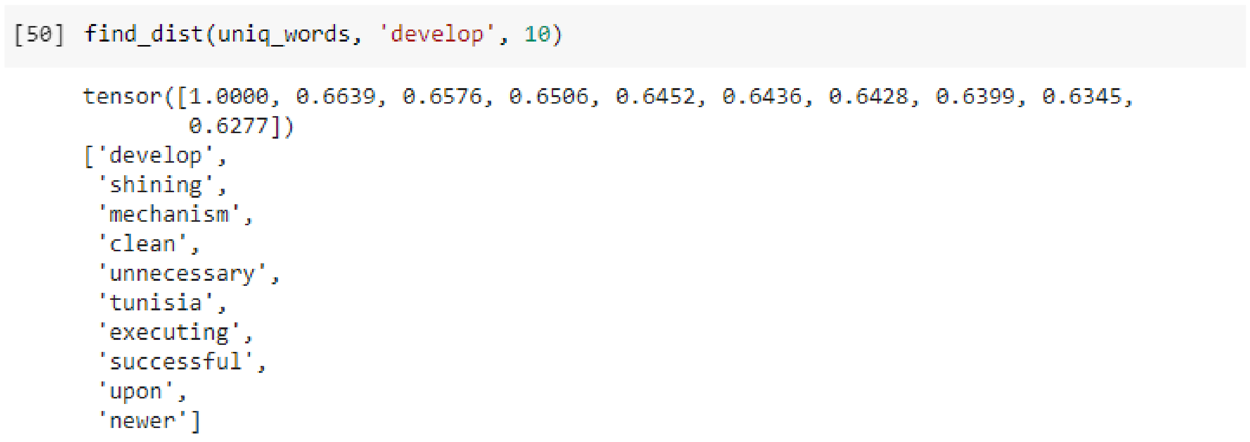
Now I have changed the window size to 2 and respective output is attached below.



```
[48] find_dist(uniq_words, 'prime', 10)

     tensor([1.0000, 0.7824, 0.6576, 0.5876, 0.5871, 0.5843, 0.5785, 0.5697, 0.5670,
             0.5661])
     ['prime',
      'minister',
      'saluted',
      'maharashtra',
      'rich',
      'indulging',
      'spiritual',
      'bj',
      'occasion',
      'medical']
```



```
[49] find_dist(uniq_words, 'president', 10)

     tensor([1.0000, 0.8288, 0.8194, 0.7697, 0.7630, 0.6891, 0.6882, 0.6512, 0.6435,
             0.6354])
     ['president',
      'tomorrow',
      'sochi',
      'former',
      'russian',
      'prior',
      'putin',
      'huge',
      'electricity',
      'asserted']
```



```
[50] find_dist(uniq_words, 'develop', 10)

     tensor([1.0000, 0.6639, 0.6576, 0.6506, 0.6452, 0.6436, 0.6428, 0.6399, 0.6345,
             0.6277])
     ['develop',
      'shining',
      'mechanism',
      'clean',
      'unnecessary',
      'tunisia',
      'executing',
      'successful',
      'upon',
      'newer']
```

Now by looking at the outputs I observe that changing window size also change the embedding vector of each words in the vocabulary.

I also observe that smaller window sizes (2) lead to embeddings where high similarity scores between two embeddings shows that the words are *interchangeable.* Furthermore, smaller window size will result the embedding vector of words which are mostly nearer to each other.

While, Larger window sizes (5) lead to embeddings where similarity is more indicative of *relatedness* of the words. And it will result in the embedding vector of words which are semantically similar to each other rather than nearer to each other.

## Doc2Vec

### What is Doc2Vec?

In machine learning, numerical representation of text information is a difficult job. Document retrieval, online search, spam filtering, topic modelling, and other applications can all benefit from such a representation **(Shperber, 2017).** However, there aren't many effective methods for doing so. Many jobs employ the well-known but unsophisticated bag of words (BOW) technique, however the results are usually average since BOW ignores many aspects of a potentially excellent representation, such as word ordering consideration.

Doc2vec is a fantastic method. It's simple to use, produces decent results, and, as the name implies, is mainly based on word2vec. So let's start with a quick overview of word2vec.

Doc2vec's purpose is to convert a document into a numerical representation, regardless of its length. Documents, unlike words, do not have logical structures, hence another approach must be devised.

Mikilov and Le employed a basic but brilliant concept: they took the word2vec model and added another vector (Paragraph ID below), as follows:

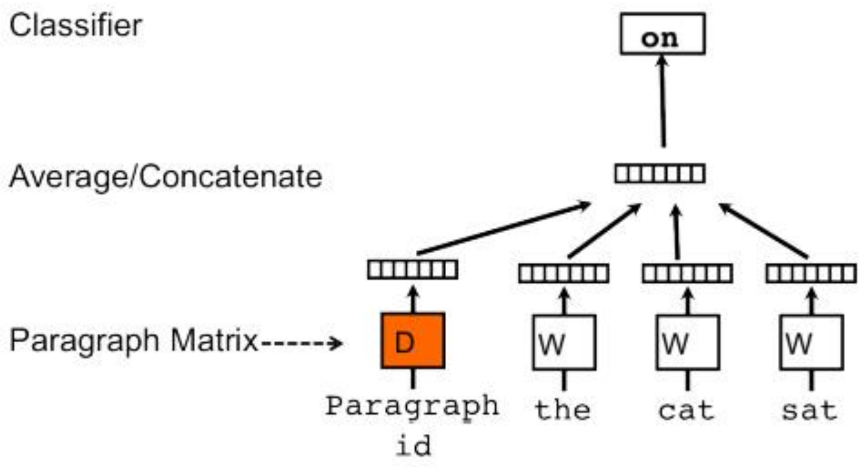


fig 3: PV-DM model

Instead of merely utilizing words to anticipate the following word, we additionally included a document-unique feature vector.

As a result, when the word vectors W are trained, the document vector D is also trained, and it carries a numeric representation of the document at the conclusion of the training.

The model described above is known as the Distributed Memory variant of the Paragraph Vector model (PV-DM). It serves as a recollection, recalling what is absent from the current context — or as the paragraph's theme.

The document vector seeks to represent the notion of a document, whereas the word vectors convey the notion of a word.

## Which Algorithm Does it Use?

We have seen that Word2Vec uses two types of algorithms which are Skip Gram and Contiouns bag of words(CBOW).Similarly Doc2Vec uses two types of alorithms which are explained below:

**Distributed Bag of Words (DBOW)**

The word2vec Skip-gram model is identical to the doc2vec DBOW model. The paragraph vectors are generated by training a neural network to predict the probability distribution of words in a paragraph based on a randomly chosen word from the paragraph.

We will vary the following parameters:

- If `dm=0`, distributed bag of words (PV-DBOW) is used; if `dm=1`,‘distributed memory’ (PV-DM) is used.
- 300- dimensional feature vectors.
- `min_count=2`, ignores all words with total frequency lower than this.
- `negative=5` , specifies how many “noise words” should be drawn.
- `hs=0` , and negative is non-zero, negative sampling will be used.
- `sample=0` , the threshold for configuring which higher-frequency words are randomly down sampled.
- `workers=cores` , use these many worker threads to train the model (=faster training with multicore machines).

**Distributed Memory (DM)**

Distributed Memory (DM) serves as a placeholder for information that isn't available in the present context — or as the topic of the paragraph. The document vector attempts to represent the concept of a document, whereas the word vectors describe the concept of a single word. We make a new Doc2Vec model with a vector size of 300 words and run it over the training corpus 30 times.

## Comparison of Doc2Vec and Word2Vec

**Similarities**

- Both Algorithm take input as a text and convert it to Numerical Vectors.
- Both word2Vec and Doc2 uses same approach for finding feature representation vector.
- Both of them use Cosine similarities for finding similarities between two words or documents.

**Differences**

- In word2vec, you train to find word vectors and then run similarity queries between words. While In doc2vec, you tag your text and you also get tag vectors
- The Word2Vec Algorithm builds distributed semantic representation of words. While In Doc2Vec you learn it for documents.
- In Word2Vec,we take one word at a time while in Doc2vec we take one document at a time.
- Word2Vec help in POS tagging while Word2doc help in sentiment analysis.

# FastText

## What is FastText?

FastText is a Word2vec extension that uses a different technique to word embedding. Instead of learning vectors for words, FastText encodes each word as an n-gram of letters. Take, for example, the word "artificial" with n=3 **(Cai, 2020).** The angle brackets denote the beginning and end of the word in the FastText representation of this term.

The embeddings can now recognise suffixes and prefixes, as well as the meaning of shorter words. After the word has been represented using character n-grams, a skip-gram model is trained to learn the embeddings. This model is known as a bag of words model with a sliding window over a word since no internal structure of the word is taken into account.

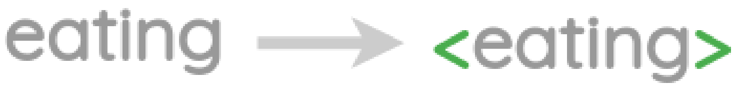
## How Does It works?

The key insight of FastText was to use the internal structure of a word to improve vector representations obtained from the skip-gram method.

Below are the steps which shows that how FastText modify Word2vec:

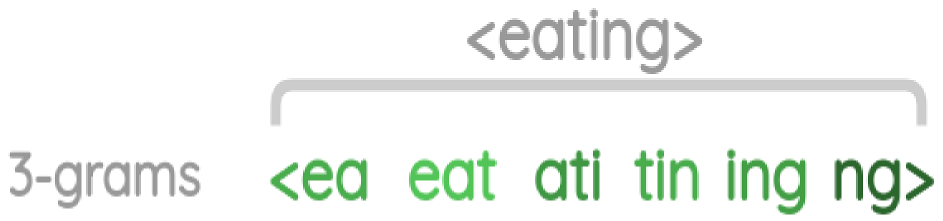
**Sub-word generation**
For a word, we generate character n-grams of length 3 to 6 present in it.

- We take a word and add angular brackets to denote the beginning and end of a word



- After that, we create character n-grams with length n. For the word "eating," for example, character n-grams of length 3 can be formed by sliding a window of three characters from the beginning of the angular bracket to the end of the angular bracket. We're going to relocate the window one step at a time here.

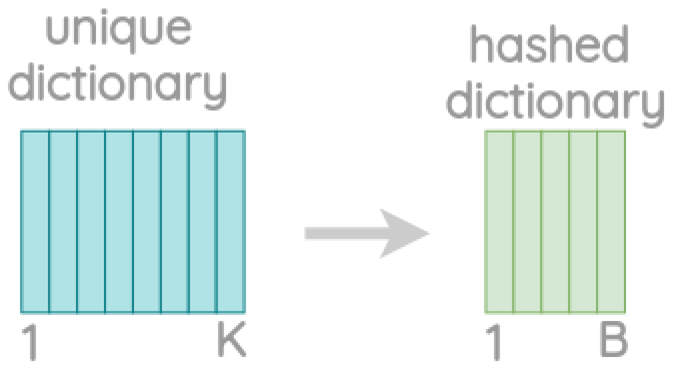
- Thus, we get a list of character n-grams for a word.



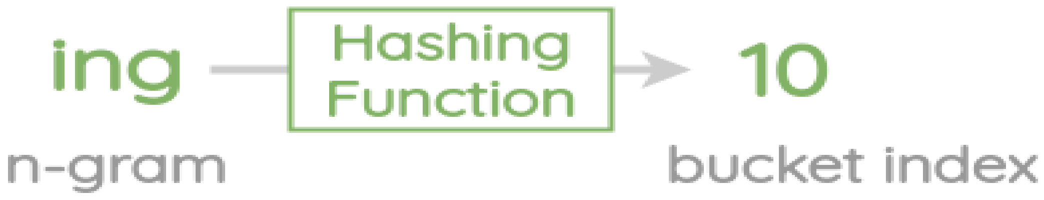
Examples of different length character n-grams are given below:

| Word | Length(n) | Character n-grams |
|---|---|---|
| eating | 3 | <ea, eat, ati, tin, ing, ng> |
| eating | 4 | <eat, eati, atin, ting, ing> |
| eating | 5 | <eati, eatin, ating, ting> |
| eating | 6 | <eatin, eating, ating> |

- We use hashing to limit the memory needs because there can be a large number of unique n-grams. Rather than learning an embedding for each every n-gram, we learn total B embeddings, with B indicating the bucket size. A bucket with a capacity of 2 million was utilised in the paper.



Each character n-gram is hashed to an integer between 1 to B. Though this could result in collisions, it helps control the vocabulary size. The paper uses the FNV-1a variant of the Fowler-Noll-Vo hashing function to hash character sequences to integer values.

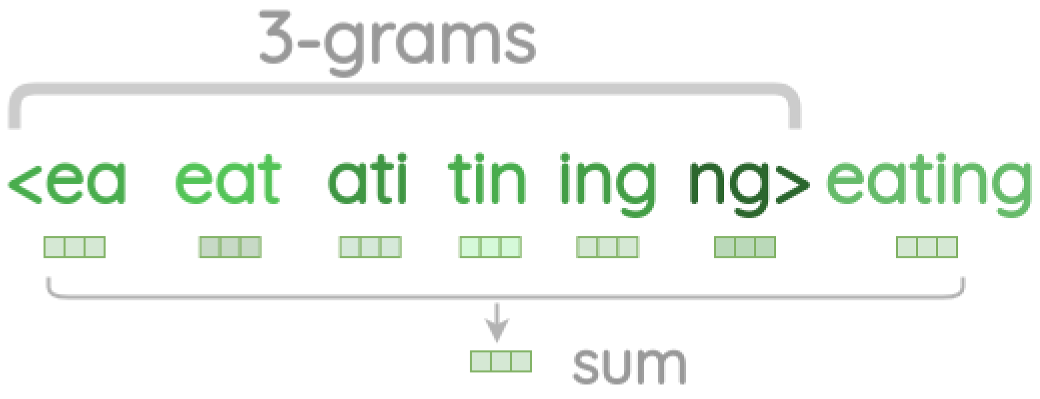


**Skip-gram with negative sampling**

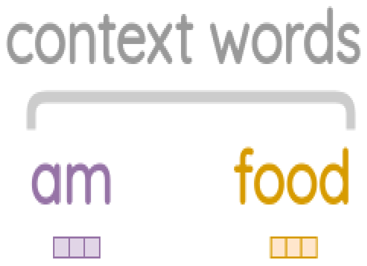
Let's look at a simple toy as an example of pre-training. We have a sentence with the word "eating" in the middle, and we need to anticipate the words "am" and "food" in the context.

I am eating food now

1. First, the centre word's embedding is determined by adding the vectors for the character n-grams and the entire word.



2. We take the real context words' word vectors directly from the embedding table, omitting the character n-grams.

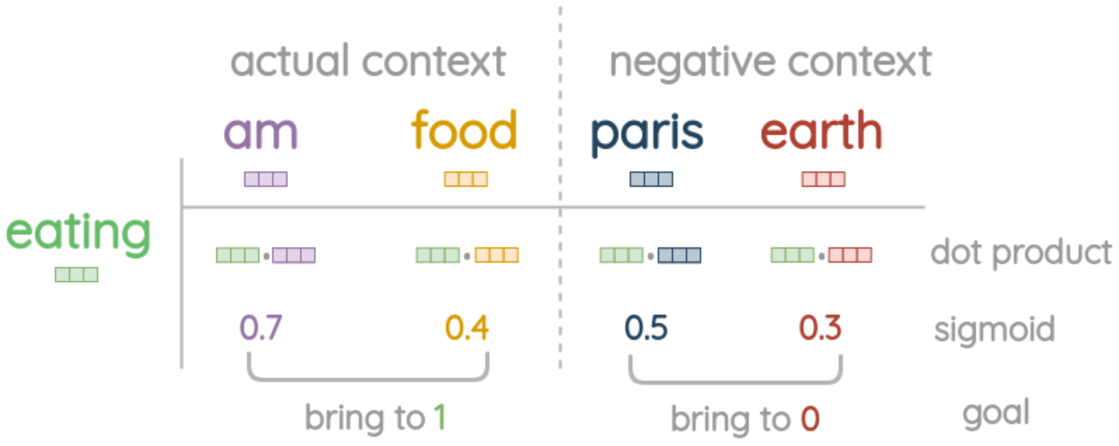


3. We now randomly gather negative samples with a probability proportional to the square root of the unigram frequency. 5 random negative words are picked for each genuine context word.

negative samples
paris
earth

4. We apply the sigmoid function to the dot product between the centre word and the real context words to produce a match score between 0 and 1.

5. Using the SGD optimizer, we change the embedding vectors based on the loss to bring true context words closer to the centre word while increasing distance to negative samples.



## How Does It Solve the problem of unseen words?

This works as long as each word in your new machine learning assignment has an embedding. However, domain-specific keywords such as product names and technical terminology are frequently lacking from the training corpus. In this situation, you have a few bad choices.

As long as each word in your new machine learning assignment has an embedding, this method will work. Domain-specific keywords such as product names and technical jargon, on the other hand, are typically absent from the training corpus. You have a few awful options in this case.

Another alternative is to search for synonyms with word embeddings using something like NLTK's Wordnet. However, this, too, misses a lot of cases and adds to the computation time and complexity.

In an ideal world, it would be possible to swiftly build appropriate embeddings for novel, out of-vocabulary ideas. FastText, a tool for learning word embeddings and sentence categorization, was just announced by Facebook Research. It can also construct embeddings for terms that aren't yet in the lexicon **(Chaudhary, 2013).**

It does so by learning vectors for letter n-grams within the word and then adding those vectors together to create the final vector or embedding for the word. Because it sees words as a sum of pieces, it can predict representations for new words by simply adding the vectors for the character n-grams it recognises in the new word.

This also helps it understand the meaning similarities between the words "rare" and "scarce." The n-gram rare sounds like rare, and the suffix -ity sounds like –ness (Reed, 2017). Because words are divided up into bits like this, it can mimic the similarity between these two words based on the similarity of their sub-grams.even if the precise words themselves have never been met in the training set — or not frequently enough to derive reliable representations.

## References

*word embedding*. (2017). Machinelearningmastery.Com. https://machinelearningmastery.com/what-are-word-embeddings/

*word embedding*. (2020). Www.Kdnuggets.Com. https://www.kdnuggets.com/2019/02/word-embeddings-nlp-applications.html

*importance of word embedding*. (2019). Hackernoon.Com. https://hackernoon.com/word-embeddings-in-nlp-and-its-applications-fab15eaf7430

*word2vec mechanism*. (2020). Wiki.Pathmind.Com. https://wiki.pathmind.com/word2vec
*skip gram*. (2019). Towarddatascience. https://towardsdatascience.com/skip-gram-nlp-context-words-prediction-algorithm-5bbf34f84e0c
*skip gram*. (2019b). Medium.Com. https://medium.datadriveninvestor.com/word2vec-skip-gram-model-explained-383fa6ddc4ae

*Word2Vec*. (2020). /Towardsdatascience.Com. https://towardsdatascience.com/implementing-word2vec-in-pytorch-skip-gram-model-e6bae040d2fb

*Word2Vec*. (2019). Medium.Com. https://medium.com/towards-datascience/word2vec-negative-sampling-made-easy-7a1a647e07a4

*Negative Sampling*. (2018). Medium.Com. https://medium.com/nearist-ai/word2vec-tutorial-part-2-negative-sampling-fd99420a6dc

*FastText*. (2020). Cai.Tools.Sap. https://cai.tools.sap/blog/glove-and-fasttext-two-popular-word-vector-models-in-nlp/

*FastText*. (2013). Amitness.Com. https://amitness.com/2020/06/fasttext-embeddings/
*FastText*. (2017). Medium.Com. https://medium.com/cisco-emerge/creating-semantic-representations-of-out-of-vocabulary-words-for-common-nlp-tasks-842dbdafba18
*Word Embedding*. (2018). /Towardsdatascience.Com. https://towardsdatascience.com/introduction-to-word-embedding-and-word2vec-652d0c2060fa