## What are Word Embeddings?

Word Embeddings are the texts converted into numbers and there may be different numerical representations of the same text.
**” Word Embeddings are Word converted into numbers ”**

**A *dictionary* may be the list of all unique words in the sentence. So, a dictionary may look like –**
**[‘Word’,’Embeddings’,’are’,’Converted’,’into’,’numbers’]**

**A ve*ctor* representation of a word may be a one-hot encoded vector where 1 stands for the position where the word exists and 0 everywhere else. The vector representation of “numbers” in this format according to the above dictionary is [0,0,0,0,0,1] and of converted is[0,0,0,1,0,0].**

**This is just a very simple method to represent a word in the vector form**

## Different types of Word Embeddings

**The different types of word embeddings can be broadly classified into two categories-**

1. **Frequency based Embedding**
2. **Prediction based Embedding**

## Frequency based Embedding

**There are generally three types of vectors that we encounter under this category.**

1. **Count Vector**
2. **TF-IDF Vector**
3. **Co-Occurrence Vector**

## Count Vector

**Consider a Corpus C of D documents {d1,d2…..dD} and N unique tokens extracted out of the corpus C. The N tokens will form our dictionary and the size of the Count Vector matrix M will be given by D X N. Each row in the matrix M contains the frequency of tokens in document D(i)**

**D1: He is a lazy boy. She is also lazy.**

**D2: Neeraj is a lazy person.**

**The dictionary created may be a list of unique tokens(words) in the corpus =['He','She','lazy','boy','Neeraj','person']**

**Here, D=2, N=6**

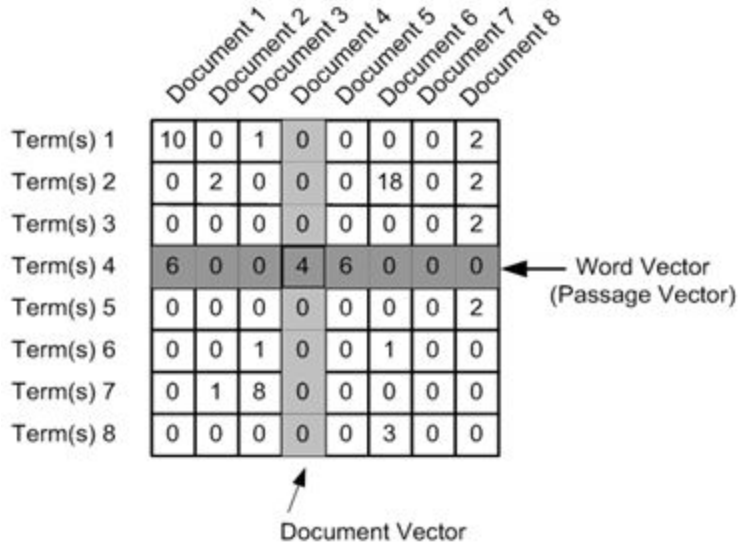
**The count matrix M of size 2 X 6 will be represented as –**

| | He | She | lazy | boy | Neera | perso |
|---|---|---|---|---|---|---|
| D1 | 1 | 1 | 2 | 1 | 0 | 0 |
| D2 | 0 | 0 | 1 | 0 | 1 | 1 |

**Now, a column can also be understood as word vector for the corresponding word in the matrix M. For example, the word vector for 'lazy' in the above matrix is [2,1] and so on.Here, the *rows* correspond to the *documents* in the corpus and the *columns* correspond to the *tokens* in the dictionary. The second row in the above matrix may be read as – D2 contains 'lazy': once, 'Neeraj': once and 'person' once.**

**Now there may be quite a few variations while preparing the above matrix M. The variations will be generally in-**

1. **The way dictionary is prepared.**
   **Why? Because in real world applications we might have a corpus which contains millions of documents. And with millions of document, we can extract hundreds of millions of unique words. So basically, the matrix that will be prepared like above will be a very sparse one and inefficient for any computation. So an alternative to using every unique word as a dictionary element would be to pick say top 10,000 words based on frequency and then prepare a dictionary.**
2. **The way count is taken for each word.**
   **We may either take the frequency (number of times a word has appeared in the document) or the presence(has the word appeared in the document?) to be the entry in the count matrix M. But generally, frequency method is preferred over the latter.**




## TF-IDF vectorization

**This is another method which is based on the frequency method but it is different to the count vectorization in the sense that it takes into account not just the occurrence of a word in a single document but in the entire corpus**

**Common words like ‘is’, ‘the’, ‘a’ etc. tend to appear quite frequently in comparison to the words which are important to a document. For example, a document A on Lionel Messi is going to contain more occurences of the word “Messi” in comparison to other documents. But common words like “the” etc. are also going to be present in higher frequency in almost every document**

**Ideally, what we would want is to down weight the common words occurring in almost all documents and give more importance to words that appear in a subset of documents**

**TF-IDF works by penalising these common words by assigning them lower weights while giving importance to words like Messi in a particular document.**

**TF = (Number of times term t appears in a document)/(Number of terms in the document)**

**IDF = log(N/n), where, N is the number of documents and n is the number of documents a term t has appeared in.**

**So, how do we explain the reasoning behind IDF? Ideally, if a word has appeared in all the document, then probably that word is not relevant to a particular document. But if it has appeared in a subset of documents then probably the word is of some relevance to the documents it is present in.**

## Co-Occurrence Matrix with a fixed context window

**The big idea** – Similar words tend to occur together and will have similar context for example – Apple is a fruit. Mango is a fruit.

Apple and mango tend to have a similar context i.e fruit.

Co-occurrence – For a given corpus, the co-occurrence of a pair of words say w1 and w2 is the number of times they have appeared together in a Context Window

Context window is specified by a number and the direction

Corpus = He is not lazy. He is intelligent. He is smart.

| | He | is | not | lazy | intelligent | smart |
|---|---|---|---|---|---|---|
| He | 0 | 4 | 2 | 1 | 2 | 1 |
| is | 4 | 0 | 1 | 2 | 2 | 1 |
| not | 2 | 1 | 0 | 1 | 0 | 0 |
| lazy | 1 | 2 | 1 | 0 | 0 | 0 |
| intelligent | 2 | 2 | 0 | 0 | 0 | 0 |
| smart | 1 | 1 | 0 | 0 | 0 | 0 |

**Variations of Co-occurrence Matrix**

**Let's say there are V unique words in the corpus. So Vocabulary size = V. The columns of the Co-occurrence matrix form the *context word*s. The different variations of Co-Occurrence Matrix are-**

1. **A co-occurrence matrix of size V X V. Now, for even a decent corpus V gets very large xand difficult to handle. So generally, this architecture is never preferred in practice.**
2. **A co-occurrence matrix of size V X N where N is a subset of V and can be obtained by removing irrelevant words like stopwords etc. for example. This is still very large and presents computational difficulties.**

**But, remember this co-occurrence matrix is not the word vector representation that is generally used. Instead, this Co-occurrence matrix is decomposed using techniques like PCA, SVD etc. into factors and combination of these factors forms the word vector representation.**

**And, a single word, instead of being represented in V dimensions will be represented in k dimensions while still capturing almost the same semantic meaning. k is generally of the order of hundreds.**

**So, what SVD does at the back is decompose Co-Occurrence matrix into three matrices, U,S and V where U and V are both orthogonal matrices. What is of importance is that dot product of U and S gives the word vector representation and V gives the word context representation.**

$$\underset{m \times n}{\overset{\hat{X}}{\begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & \\ \vdots & \vdots & \ddots & \\ x_{m1} & & & x_{mn} \end{pmatrix}}} \approx \underset{m \times r}{\overset{U}{\begin{pmatrix} u_{11} & \cdots & u_{1r} \\ \vdots & \ddots & \\ u_{m1} & & u_{mr} \end{pmatrix}}} \underset{r \times r}{\overset{S}{\begin{pmatrix} s_{11} & 0 & \cdots \\ 0 & \ddots & \\ \vdots & & s_{rr} \end{pmatrix}}} \underset{r \times n}{\overset{V^{\mathsf{T}}}{\begin{pmatrix} v_{11} & \cdots & v_{1n} \\ \vdots & \ddots & \\ v_{r1} & & v_{rn} \end{pmatrix}}}$$

## Advantages of Co-occurrence Matrix

1. **It preserves the semantic relationship between words. i.e man and woman tend to be closer than man and apple.**
2. **It uses SVD at its core, which produces more accurate word vector representations than existing methods.**
3. **It uses factorization which is a well-defined problem and can be efficiently solved.**
4. **It has to be computed once and can be used anytime once computed. In this sense, it is faster in comparison to others.**

## Disadvantages of Co-Occurrence Matrix

1. **It requires huge memory to store the co-occurrence matrix.**
   **But, this problem can be circumvented by factorizing the matrix out of the system for example in Hadoop clusters etc. and can be saved**

## Prediction based Vector

**So far, we have seen deterministic methods to determine word vectors**

**These methods were prediction based in the sense that they provided probabilities to the words and proved to be state of the art for tasks like word analogies and word similarities.**

**Word2vec is not a single algorithm but a combination of two techniques – CBOW(Continuous bag of words) and Skip-gram model. Both of these are shallow neural networks which map word(s) to the target variable which is also a word(s). Both of these techniques learn weights which act as word vector representations.**

**CBOW (Continuous Bag of words):**

**The way CBOW work is that it tends to predict the probability of a word given a context. A context may be a single word or a group of words**

**Suppose, we have a corpus C = “Hey, this is sample corpus using only one context word.” and we have defined a context window of 1.**

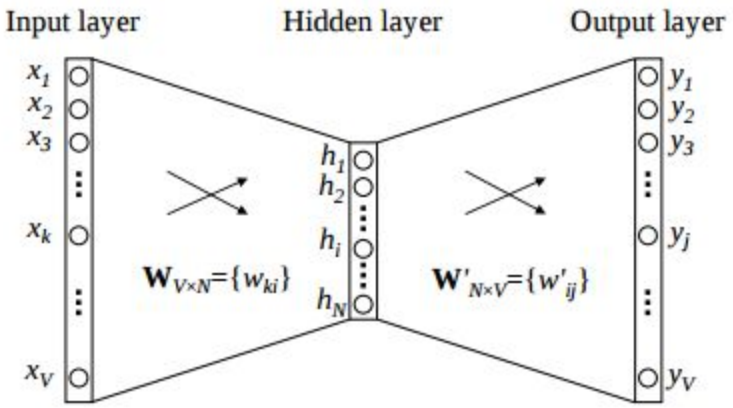
**This corpus may be converted into a training set for a CBOW model as follow.**

| Input | Output | | | Hey | This | is | sample | corpus | using | only | one | context | word |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Hey | this | | Datapoint 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| this | hey | | Datapoint 2 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| is | this | | Datapoint 3 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| is | sample | | Datapoint 4 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| sample | is | | Datapoint 5 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| sample | corpus | | Datapoint 6 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| corpus | sample | | Datapoint 7 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| corpus | using | | Datapoint 8 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| using | corpus | | Datapoint 9 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| using | only | | Datapoint 10 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| only | using | | Datapoint 11 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| only | one | | Datapoint 12 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| one | only | | Datapoint 13 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| one | context | | Datapoint 14 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| context | one | | Datapoint 15 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| context | word | | Datapoint 16 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| word | context | | Datapoint 17 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

**The target for a single datapoint say Datapoint 4 is shown as below**

| Hey | this | is | sample | corpus | using | only | one | context | word |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |

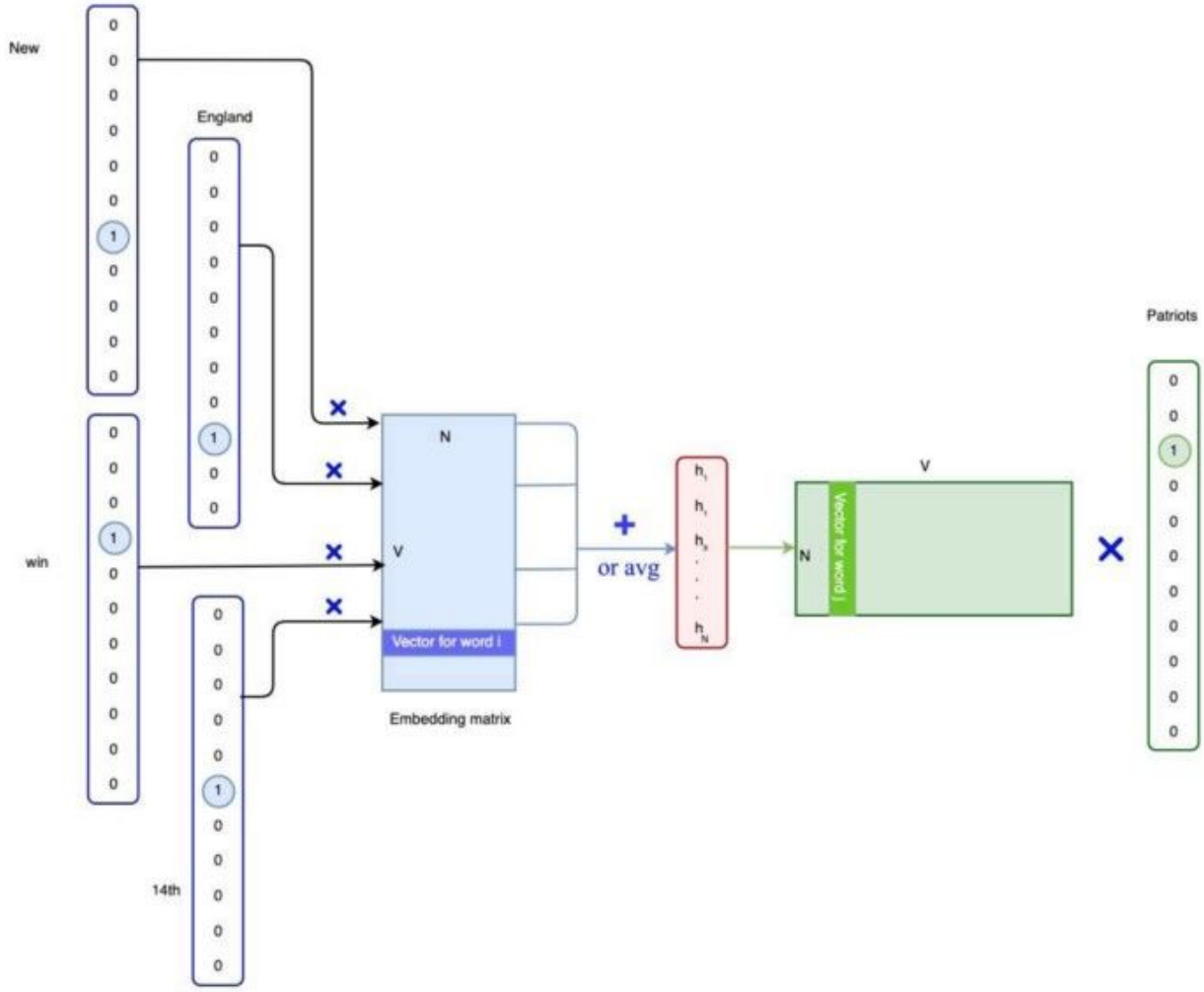
**This matrix shown above is sent into a shallow neural network with three layers: an input layer, a hidden layer and an output layer. The output layer is a softmax layer which is used to sum the probabilities obtained in the output layer to 1.**

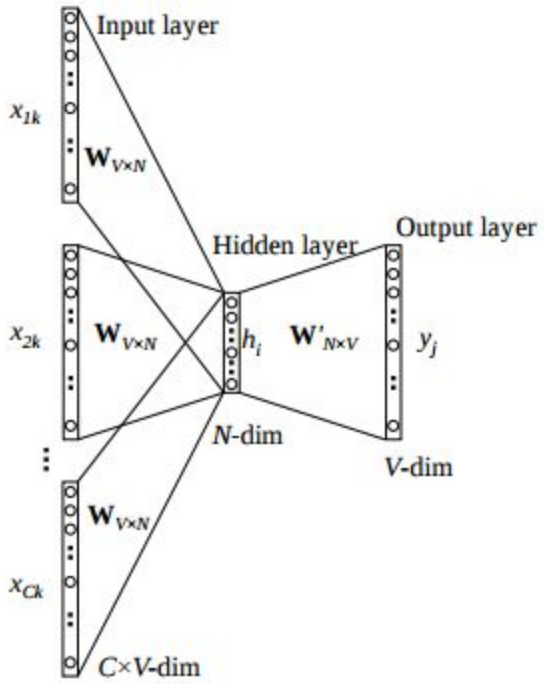



**The flow is as follows:**

1. **The input layer and the target, both are one- hot encoded of size [1 X V]. Here V=10 in the above example.**
2. **There are two sets of weights. one is between the input and the hidden layer and second between hidden and output layer. Input-Hidden layer matrix size =[V X N] , hidden-Output layer matrix size =[N X V] : Where N is the number of dimensions we choose to represent our word in. It is arbitary and a hyper-parameter for a Neural Network. Also, N is the number of neurons in the hidden layer. Here, N=4.**
3. **There is a no activation function between any layers.( More specifically, I am referring to linear activation)**
4. **The input is multiplied by the input-hidden weights and called hidden activation. It is simply the corresponding row in the input-hidden matrix copied.**
5. **The hidden input gets multiplied by hidden- output weights and output is calculated.**
6. **Error between output and target is calculated and propagated back to re-adjust the weights.**
7. **The weight  between the hidden layer and the output layer is taken as the word vector representation of the word.**

**We saw the above steps for a single context word. Now, what about if we have multiple context words? The image below describes the architecture for multiple context words.**


New
England
win
14th
N
V
Vector for word i
Embedding matrix
+
or avg
h1
h2
h3
hN
V
N
Vector for word j
Patriots




Context

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| C1 | this | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| C2 | corpus | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| C3 | context | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

Input-Hidden Weight

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | 16 |
| 17 | 18 | 19 | 20 |
| 21 | 22 | 23 | 24 |
| 25 | 26 | 27 | 28 |
| 29 | 30 | 31 | 32 |
| 33 | 34 | 35 | 36 |
| 37 | 38 | 39 | 40 |

Hidden Activation

| | | | |
|---|---|---|---|
| 5 | 6 | 7 | 8 |
| 17 | 18 | 19 | 20 |
| 33 | 34 | 35 | 36 |

Average hidden Activation

| | | | |
|---|---|---|---|
| 18.33333333 | 19.33333333 | 20.33333333 | 21.33333333 |

**The image above takes 3 context words and predicts the probability of a target word. The input can be assumed as taking three one-hot encoded vectors in the input layer as shown above in red, blue and green.**

**So, the input layer will have 3 [1 X V] Vectors in the input as shown above and 1 [1 X V] in the output layer. Rest of the architecture is same as for a 1-context CBOW.**

**The steps remain the same, only the calculation of hidden activation changes. Instead of just copying the corresponding rows of the input-hidden weight matrix to the hidden layer, an average is taken over all the corresponding rows of the matrix**

**So, if we have three context words for a single target word, we will have three initial hidden activations which are then averaged element-wise to obtain the final activation**

**1.) We initialize weights and biases with random values (This is one time initiation. In the next iteration, we will use updated weights, and biases). Let us define:**

- **wh as weight matrix to the hidden layer**
- **bh as bias matrix to the hidden layer**
- **wout as weight matrix to the output layer**
- **bout as bias matrix to the output layer**

**Step 0: Read input and output**

| X | | | | wh | | | bh | | | hidden_layer_input | | | hidden_layer_activations | | | wout | bout | output | y | E |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | | | | | | | | | | | | | | | | 1 | |
| 1 | 0 | 1 | 1 | | | | | | | | | | | | | | | | 1 | |
| 0 | 1 | 0 | 1 | | | | | | | | | | | | | | | | 0 | |

**Step 1: Initialize weights and biases with random values (There are methods to initialize weights and biases but for now initialize with random values)**

| X | | | | wh | | | bh | | | hidden_layer_input | | | hidden_layer_activations | | | wout | bout | output | y | E |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0.42 | 0.88 | 0.55 | 0.46 | 0.72 | 0.08 | | | | | | | 0.30 | 0.69 | | 1 | |
| 1 | 0 | 1 | 1 | 0.10 | 0.73 | 0.68 | | | | | | | | | | 0.25 | | | 1 | |
| 0 | 1 | 0 | 1 | 0.60 | 0.18 | 0.47 | | | | | | | | | | 0.23 | | | 0 | |
| | | | | 0.92 | 0.11 | 0.52 | | | | | | | | | | | | | | |

**Step 2: Calculate hidden layer input:**

**hidden_layer_input= matrix_dot_product(X,wh) + bh**

| X | | | | wh | | | bh | | | hidden_layer_input | | | hidden_layer_activations | | | wout | bout | output | y | E |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0.42 | 0.88 | 0.55 | 0.46 | 0.72 | 0.08 | 1.48 | 1.78 | 1.10 | | | | 0.30 | 0.69 | | 1 | |
| 1 | 0 | 1 | 1 | 0.10 | 0.73 | 0.68 | | | | 2.40 | 1.89 | 1.61 | | | | 0.25 | | | 1 | |
| 0 | 1 | 0 | 1 | 0.60 | 0.18 | 0.47 | | | | 1.48 | 1.56 | 1.27 | | | | 0.23 | | | 0 | |
| | | | | 0.92 | 0.11 | 0.52 | | | | | | | | | | | | | | |

**Step 3:** Perform non-linear transformation on hidden linear input

**IN CASE OF CBOW MODEL WE DO NOT PERFORM ANY SUCH NON LINEAR TRANSFORMATION.**

***hiddenlayer_activations = sigmoid(hidden_layer_input)***

| X | | | | wh | | | bh | | | hidden_layer_input | | | hidden_layer_activations | | | wout | bout | output | y | E |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0.42 | 0.88 | 0.55 | 0.46 | 0.72 | 0.08 | 1.48 | 1.78 | 1.10 | 0.81 | 0.86 | 0.75 | 0.30 | 0.69 | | 1 | |
| 1 | 0 | 1 | 1 | 0.10 | 0.73 | 0.68 | | | | 2.40 | 1.89 | 1.61 | 0.92 | 0.87 | 0.83 | 0.25 | | | 1 | |
| 0 | 1 | 0 | 1 | 0.60 | 0.18 | 0.47 | | | | 1.48 | 1.56 | 1.27 | 0.81 | 0.83 | 0.78 | 0.23 | | | 0 | |
| | | | | 0.92 | 0.11 | 0.52 | | | | | | | | | | | | | | |

**Step 4:** Perform linear and non-linear transformation of hidden layer activation at output layer

**output_layer_input = matrix_dot_product (hiddenlayer_activations * wout ) + bout**

***output = sigmoid(output_layer_input)***

| X | | | | wh | | | bh | | | hidden_layer_input | | | hidden_layer_activations | | | wout | bout | output | y | E |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0.42 | 0.88 | 0.55 | 0.46 | 0.72 | 0.08 | 1.48 | 1.78 | 1.10 | 0.81 | 0.86 | 0.75 | 0.30 | 0.69 | 0.79 | 1 | |
| 1 | 0 | 1 | 1 | 0.10 | 0.73 | 0.68 | | | | 2.40 | 1.89 | 1.61 | 0.92 | 0.87 | 0.83 | 0.25 | | 0.80 | 1 | |
| 0 | 1 | 0 | 1 | 0.60 | 0.18 | 0.47 | | | | 1.48 | 1.56 | 1.27 | 0.81 | 0.83 | 0.78 | 0.23 | | 0.79 | 0 | |
| | | | | 0.92 | 0.11 | 0.52 | | | | | | | | | | | | | | |

## Step 5: Calculate gradient of Error(E) at output layer

***E = y-output***

| X | | | | wh | | | bh | | | hidden_layer_input | | | hidden_layer_activations | | | wout | bout | output | y | E |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0.42 | 0.88 | 0.55 | 0.46 | 0.72 | 0.08 | 1.48 | 1.78 | 1.10 | 0.81 | 0.86 | 0.75 | 0.30 | 0.69 | 0.79 | 1 | 0.21 |
| 1 | 0 | 1 | 1 | 0.10 | 0.73 | 0.68 | | | | 2.40 | 1.89 | 1.61 | 0.92 | 0.87 | 0.83 | 0.25 | | 0.80 | 1 | 0.20 |
| 0 | 1 | 0 | 1 | 0.60 | 0.18 | 0.47 | | | | 1.48 | 1.56 | 1.27 | 0.81 | 0.83 | 0.78 | 0.23 | | 0.79 | 0 | -0.79 |
| | | | | 0.92 | 0.11 | 0.52 | | | | | | | | | | | | | | |

## Step 6: Compute slope at output and hidden layer

***Slope_output_layer= derivatives_sigmoid(output)***

***Slope_hidden_layer = derivatives_sigmoid(hiddenlayer_activations)***

| X | | | | wh | | | bh | | | hidden_layer_input | | | hidden_layer_activations | | | wout | bout | output | y | E |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0.42 | 0.88 | 0.55 | 0.46 | 0.72 | 0.08 | 1.48 | 1.78 | 1.10 | 0.81 | 0.86 | 0.75 | 0.30 | 0.69 | 0.79 | 1 | 0.21 |
| 1 | 0 | 1 | 1 | 0.10 | 0.73 | 0.68 | | | | 2.40 | 1.89 | 1.61 | 0.92 | 0.87 | 0.83 | 0.25 | | 0.80 | 1 | 0.20 |
| 0 | 1 | 0 | 1 | 0.60 | 0.18 | 0.47 | | | | 1.48 | 1.56 | 1.27 | 0.81 | 0.83 | 0.78 | 0.23 | | 0.79 | 0 | -0.79 |
| | | | | 0.92 | 0.11 | 0.52 | | | | | | | | | | | | | | |

| Slope hidden layer | | |
|---|---|---|
| 0.15 | 0.12 | 0.19 |
| 0.08 | 0.11 | 0.14 |
| 0.15 | 0.14 | 0.17 |

| Slope Output |
|---|
| 0.17 |
| 0.16 |
| 0.17 |

## Step 7: Compute delta at output layer

### *d_output = E * slope_output_layer*lr*

| X | | | | wh | | | bh | | | hidden_layer_input | | | hidden_layer_activations | | | wout | bout | output | y | E |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0.42 | 0.88 | 0.55 | 0.46 | 0.72 | 0.08 | 1.48 | 1.78 | 1.10 | 0.81 | 0.86 | 0.75 | 0.30 | 0.69 | 0.79 | 1 | 0.21 |
| 1 | 0 | 1 | 1 | 0.10 | 0.73 | 0.68 | | | | 2.40 | 1.89 | 1.61 | 0.92 | 0.87 | 0.83 | 0.25 | | 0.80 | 1 | 0.20 |
| 0 | 1 | 0 | 1 | 0.60 | 0.18 | 0.47 | | | | 1.48 | 1.56 | 1.27 | 0.81 | 0.83 | 0.78 | 0.23 | | 0.79 | 0 | -0.79 |
| | | | | 0.92 | 0.11 | 0.52 | | | | | | | | | | | | | | |

| Slope hidden layer | | |
|---|---|---|
| 0.15 | 0.12 | 0.19 |
| 0.08 | 0.11 | 0.14 |
| 0.15 | 0.14 | 0.17 |

| Slope Output |
|---|
| 0.17 |
| 0.16 |
| 0.17 |

| E |
|---|
| 0.21 |
| 0.20 |
| -0.79 |

| delta output |
|---|
| 0.04 |
| 0.03 |
| -0.13 |

## Step 8: Calculate Error at hidden layer

### Error_at_hidden_layer = matrix_dot_product(d_output, wout.Transpose)

| X | | | | wh | | | bh | | | hidden_layer_input | | | hidden_layer_activations | | | wout | bout | output | y | E |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0.42 | 0.88 | 0.55 | 0.46 | 0.72 | 0.08 | 1.48 | 1.78 | 1.10 | 0.81 | 0.86 | 0.75 | 0.30 | 0.69 | 0.79 | 1 | 0.21 |
| 1 | 0 | 1 | 1 | 0.10 | 0.73 | 0.68 | | | | 2.40 | 1.89 | 1.61 | 0.92 | 0.87 | 0.83 | 0.25 | | 0.80 | 1 | 0.20 |
| 0 | 1 | 0 | 1 | 0.60 | 0.18 | 0.47 | | | | 1.48 | 1.56 | 1.27 | 0.81 | 0.83 | 0.78 | 0.23 | | 0.79 | 0 | -0.79 |
| | | | | 0.92 | 0.11 | 0.52 | | | | | | | | | | | | | | |

| Slope hidden layer | | | error at hidden layer | | |
|---|---|---|---|---|---|
| 0.15 | 0.12 | 0.19 | 0.010 | 0.009 | 0.008 |
| 0.08 | 0.11 | 0.14 | 0.010 | 0.008 | 0.008 |
| 0.15 | 0.14 | 0.17 | -0.039 | -0.033 | -0.031 |

| Slope Output |
|---|
| 0.17 |
| 0.16 |
| 0.17 |

| E |
|---|
| 0.21 |
| 0.20 |
| -0.79 |

| delta output |
|---|
| 0.04 |
| 0.03 |
| -0.13 |

## Step 9: Compute delta at hidden layer

***d_hiddenlayer = Error_at_hidden_layer * slope_hidden_layer***

| X | | | | wh | | | bh | | | hidden_layer_input | | | hidden_layer_activations | | | wout | bout | output | y | E |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0.42 | 0.88 | 0.55 | 0.46 | 0.72 | 0.08 | 1.48 | 1.78 | 1.10 | 0.81 | 0.86 | 0.75 | 0.30 | 0.69 | 0.79 | 1 | 0.21 |
| 1 | 0 | 1 | 1 | 0.10 | 0.73 | 0.68 | | | | 2.40 | 1.89 | 1.61 | 0.92 | 0.87 | 0.83 | 0.25 | | 0.80 | 1 | 0.20 |
| 0 | 1 | 0 | 1 | 0.60 | 0.18 | 0.47 | | | | 1.48 | 1.56 | 1.27 | 0.81 | 0.83 | 0.78 | 0.23 | | 0.79 | 0 | -0.79 |
| | | | | 0.92 | 0.11 | 0.52 | | | | | | | | | | | | | | |

| Slope hidden layer | | | error at hidden layer | | |
|---|---|---|---|---|---|
| 0.15 | 0.12 | 0.19 | 0.010 | 0.009 | 0.008 |
| 0.08 | 0.11 | 0.14 | 0.010 | 0.008 | 0.008 |
| 0.15 | 0.14 | 0.17 | -0.039 | -0.033 | -0.031 |

| Slope Output |
|---|
| 0.17 |
| 0.16 |
| 0.17 |

| E |
|---|
| 0.21 |
| 0.20 |
| -0.79 |

| delta hidden layer | | |
|---|---|---|
| 0.002 | 0.001 | 0.002 |
| 0.001 | 0.001 | 0.001 |
| -0.006 | -0.005 | -0.005 |

| delta output |
|---|
| 0.04 |
| 0.03 |
| -0.13 |

## Step 10: Update weight at both output and hidden layer

**wout = wout + matrix_dot_product(hiddenlayer_activations.Transpose, d_output)*learning_rate**

**wh = wh+ matrix_dot_product(X.Transpose,d_hiddenlayer)*learning_rate**

| X | | | | wh | | | bh | | | hidden_layer_input | | | hidden_layer_activations | | | wout | bout | output | y | E |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0.42 | 0.88 | 0.55 | 0.46 | 0.72 | 0.08 | 1.48 | 1.78 | 1.10 | 0.81 | 0.86 | 0.75 | 0.29 | 0.69 | 0.79 | 1 | 0.21 |
| 1 | 0 | 1 | 1 | 0.10 | 0.73 | 0.68 | | | | 2.40 | 1.89 | 1.61 | 0.92 | 0.87 | 0.83 | 0.25 | | 0.80 | 1 | 0.20 |
| 0 | 1 | 0 | 1 | 0.60 | 0.18 | 0.47 | | | | 1.48 | 1.56 | 1.27 | 0.81 | 0.83 | 0.78 | 0.23 | | 0.79 | 0 | -0.79 |
| | | | | 0.92 | 0.11 | 0.51 | | | | | | | | | | | | | | |

| Slope hidden layer | | | error at hidden layer | | |
|---|---|---|---|---|---|
| 0.15 | 0.12 | 0.19 | 0.010 | 0.009 | 0.008 |
| 0.08 | 0.11 | 0.14 | 0.010 | 0.008 | 0.008 |
| 0.15 | 0.14 | 0.17 | -0.039 | -0.033 | -0.031 |

| Slope Output |
|---|
| 0.17 |
| 0.16 |
| 0.17 |

| E |
|---|
| 0.21 |
| 0.20 |
| -0.79 |

| Learning Rate | 0.1 |
|---|---|

| delta hidden layer | | |
|---|---|---|
| 0.002 | 0.001 | 0.002 |
| 0.001 | 0.001 | 0.001 |
| -0.006 | -0.005 | -0.005 |

| delta output |
|---|
| 0.035 |
| 0.033 |
| -0.131 |

**Step 11:** Update biases at both output and hidden layer

***bh = bh + sum(d_hiddenlayer, axis=0) * learning_rate***

***bout = bout + sum(d_output, axis=0)*learning_rate***

| X | | | | wh | | | bh | | | hidden_layer_input | | | hidden_layer_activations | | | wout | bout | output | y | E |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0.42 | 0.88 | 0.55 | 0.46 | 0.72 | 0.08 | 1.48 | 1.78 | 1.10 | 0.81 | 0.86 | 0.75 | 0.29 | 0.68 | 0.79 | 1 | 0.21 |
| 1 | 0 | 1 | 1 | 0.10 | 0.73 | 0.68 | | | | 2.40 | 1.89 | 1.61 | 0.92 | 0.87 | 0.83 | 0.25 | | 0.80 | 1 | 0.20 |
| 0 | 1 | 0 | 1 | 0.60 | 0.18 | 0.47 | | | | 1.48 | 1.56 | 1.27 | 0.81 | 0.83 | 0.78 | 0.23 | | 0.79 | 0 | -0.79 |
| | | | | 0.92 | 0.11 | 0.51 | | | | | | | | | | | | | | |

| Slope hidden layer | | | error at hidden layer | | |
|---|---|---|---|---|---|
| 0.15 | 0.12 | 0.19 | 0.010 | 0.009 | 0.008 |
| 0.08 | 0.11 | 0.14 | 0.010 | 0.008 | 0.008 |
| 0.15 | 0.14 | 0.17 | -0.039 | -0.033 | -0.031 |

| Slope Output |
|---|
| 0.17 |
| 0.16 |
| 0.17 |

| E |
|---|
| 0.21 |
| 0.20 |
| -0.79 |

| Learning Rate | 0.1 |
|---|---|

| delta hidden layer | | |
|---|---|---|
| 0.002 | 0.001 | 0.002 |
| 0.001 | 0.001 | 0.001 |
| -0.006 | -0.005 | -0.005 |

| delta output |
|---|
| 0.035 |
| 0.033 |
| -0.131 |

**Advantages of CBOW:**

1. Being probabilistic is nature, it is supposed to perform superior to deterministic methods(generally).
2. It is low on memory. It does not need to have huge RAM requirements like that of co-occurrence matrix where it needs to store three huge matrices.

**Disadvantages of CBOW:**

1. CBOW takes the average of the context of a word (as seen above in calculation of hidden activation). For example, Apple can be both a

fruit and a company but CBOW takes an average of both the contexts and places it in between a cluster for fruits and companies.

2. Training a CBOW from scratch can take forever if not properly optimized.

## Skip – Gram model

Skip – gram follows the same topology as of CBOW. It just flips CBOW's architecture on its head. The aim of skip-gram is to predict the context given a word

C="Hey, this is sample corpus using only one context word."

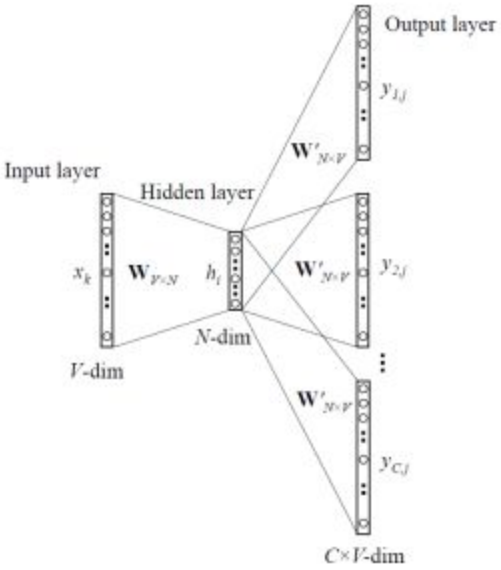
| Input | Output(Context1) | Output(Context2) |
|---|---|---|
| Hey | this | <padding> |
| this | Hey | is |
| is | this | sample |
| sample | is | corpus |
| corpus | sample | corpus |
| using | corpus | only |
| only | using | one |
| one | only | context |
| context | one | word |
| word | context | <padding> |

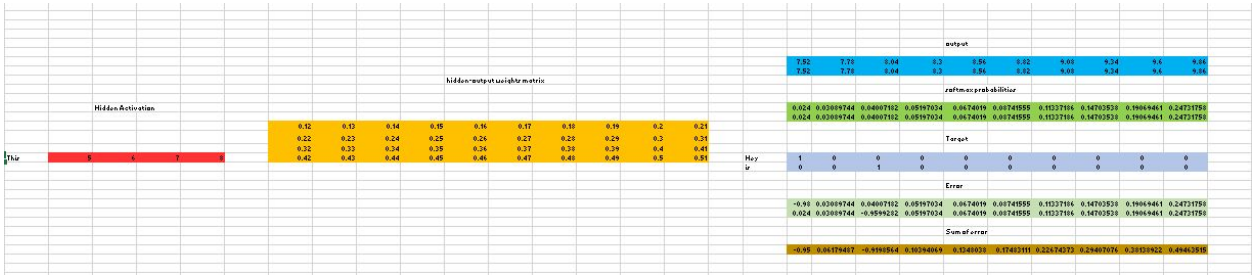
The input vector for skip-gram is going to be similar to a 1-context CBOW model. Also, the calculations up to hidden layer activations are going to be the same. The difference will be in the target variable. Since we have

**defined a context window of 1 on both the sides, there will be “two” one hot encoded target variables and “two” corresponding outputs**

**Two separate errors are calculated with respect to the two target variables and the two error vectors obtained are added element-wise to obtain a final error vector which is propagated back to update the weights.**

**The weights between the input and the hidden layer are taken as the word vector representation after training.**



**Input layer  size – [1 X V], Input hidden weight matrix size – [V X N], Number of neurons in hidden layer – N, Hidden-Output weight matrix size – [N X V], Output layer size – C [1 X V]**

**In the above example, C is the number of context words=2, V= 10, N=4**

1. **The row in red is the hidden activation corresponding to the input one-hot encoded vector. It is basically the corresponding row of input-hidden matrix copied.**
2. **The yellow matrix is the weight between the hidden layer and the output layer.**
3. **The blue matrix is obtained by the matrix multiplication of hidden activation and the hidden output weights. There will be two rows calculated for two target(context) words.**
4. **Each row of the blue matrix is converted into its softmax probabilities individually as shown in the green box.**
5. **The grey matrix contains the one hot encoded vectors of the two context words(target).**
6. **Error is calculated by substracting the first row of the grey matrix(target) from the first row of the green matrix(output) element-wise. This is repeated for the next row. Therefore, for n target context words, we will have n error vectors.**
7. **Element-wise sum is taken over all the error vectors to obtain a final error vector.**
8. **This error vector is propagated back to update the weights**

## Word2Vec

1. It is kind of a package that contains algorithms and training methods.

### Skip-Gram Model:

For each estimation step we take one word as the centre word and try and predict words in it's context out to some window size

Model is a probability distribution which defines the probability of a word appearing in the context given this centre word.

Choose vector representations of words so that we can try and maximise probability distribution.

We just have one probability distribution of a context word (output) occuring in context close to centre word.

### Details:

For each word in the entire corpus ($t = 1$ to $T$) predict surrounding words in a window of radius ($m$) of every word.

Window of radius ($m$) means ($m$) words before and after the centre word constitute the context.

loss fn:

$$J'(\theta) = \prod_{t=1}^{T} \prod_{\substack{-m \le j \le m \\ j \neq 0}} P(w_{t+j} \mid w_t ; \theta)$$

This is the loss or objective function.

what it actually means is that we are going to take a big corpus of text and we are going to go through each position in the text and for each position in the text we are going to have a window of size (2m) around it (m-before, m-after) and we are going to have a probability distribution that will give probability to a word appearing in the context of center word.

What we want is to set the parameters of this model, so that the probabilities are highest for the words that do appear in the context of centre word.

Parameter of this model — ($\theta$).
This is the vector representation of the words.
This is the only parameter in this model and it is the vector representation of each word.

Negative (log) likelihood:

$$J(\theta) = -\frac{1}{T}\sum_{t=1}^{T}\sum_{\substack{-m\le j\le m \\ j\neq 0}} \log P(w_{t+j}|w_t)$$

NOTES

This is the same objective f^n as before just transformed to make math easy.

THURSDAY

Now we have to minimise this objective fn by optimising the variable ($\theta$).

Use word vector to minimise the negative (log) likelihood.
We will have a probability distribution such that the prob. of context word is maximised.

Probability distribution:

$$P(O|C) = \frac{\exp(u_o^T \cdot v_c)}{\sum_{w=1}^{V} \exp(u_w^T \cdot v_c)}$$

$\{C, O\} \rightarrow$ Indices in the space of vocabulary

$P(w_{t+j} | w_t)$: here $(w_{t+j})$ and $(w_t)$ were positions of words in the corpus

$u_o$: word vector associated with context word in index 'o'.
$v_c$: word vector associated with the centre word.

This is the softmax form where we do the dot product of centre word vector ($V_c$) and context word vector ($U_o$) to have an idea of similarity.

General softmax form: $P_i = \frac{e^{u_i}}{\sum_j e^{u_j}}$

2017

Softmax fam is a standard way to turn numbers in a probability distribution.
The dot products are just real no.

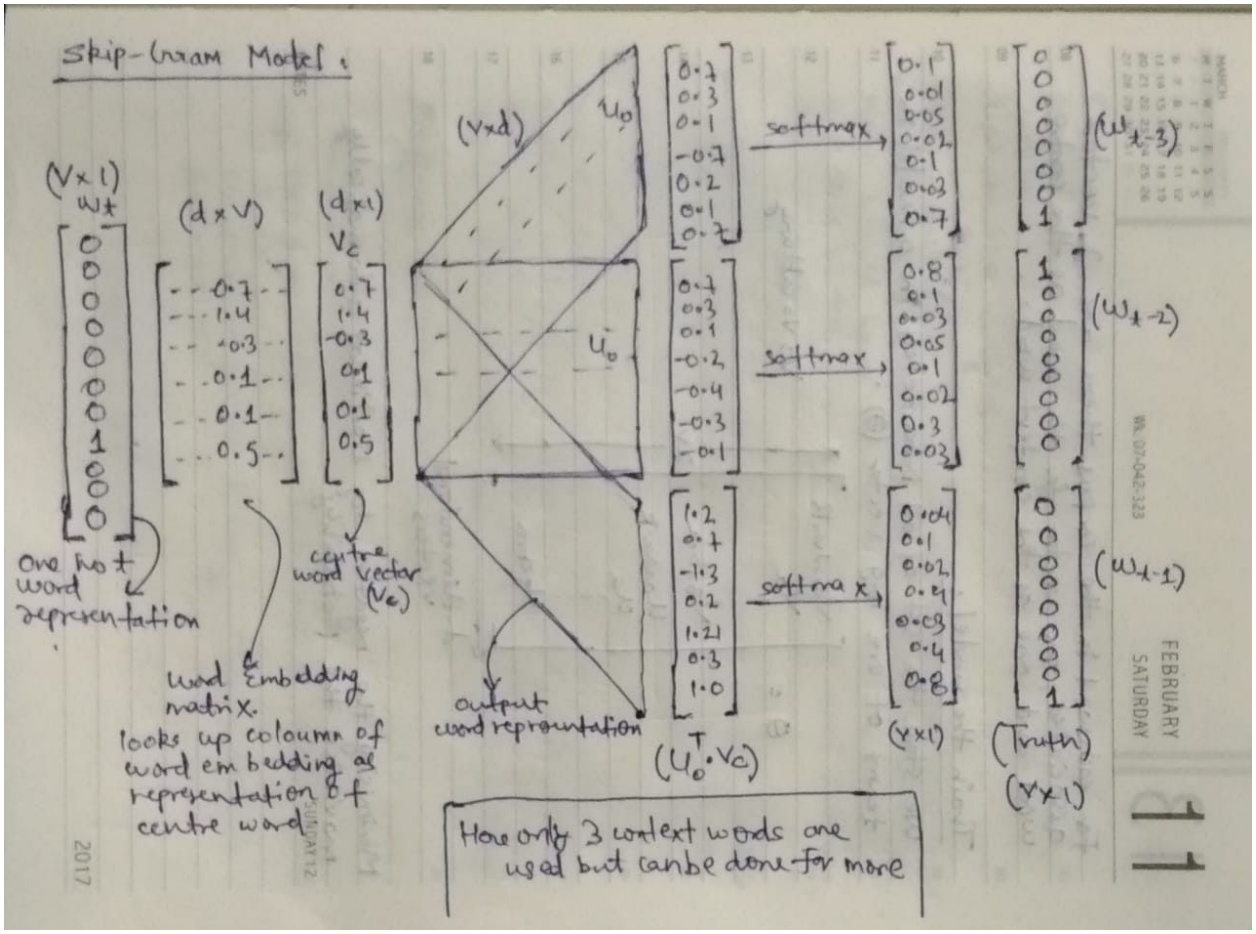
$u_o^T \cdot v_c$ = Real No.

can be (-ve) can be (+ve).

So we use exponential f^n to make them (+ve).
and once we have all no. (+ve) and we want a probability distribution.
The simplest thing to do is divide each no. by the sum of all.

$$P(o|c) \Rightarrow \frac{e^{(u_o^T \cdot v_c)}}{\sum_{w=1}^{V} e^{(u_w^T \cdot v_c)}}$$

Softmax using 'c' to give probability of 'o'.


Skip-Gram Model:
(V×1)
$w_t$
(d×V)
(d×1)
$v_c$
(V×d)
$U_o$
$U_o$
softmax
softmax
softmax
one hot word representation
word Embedding matrix.
looks up column of word embedding as representation of centre word
centre word vector ($v_c$)
output word representation
$(U_o^T \cdot v_c)$
(V×1)
(Truth)
(V×1)
$(w_{t-3})$
$(w_{t-2})$
$(w_{t-1})$
Here only 3 context words are used but can be done for more

Once we obtain the softmax probabilities for each context word, we can have the corresponding error vectors which then can be backpropagated to update the weights.

After the final iteration, the word embedding matrix which was once initially random initialized now would have learned the correct embeddings and this matrix shows the different word vectors for each word in corpus.

⟹ We just need to give the hyperparameter window size, the algorithm will start from the first word of corpus and shift one by one to the last word.

Once the algorithm moves from one centre word to another centre word updated weights or matrices are used. So by the time it will reach the last word of corpus, it would have learnt all the word embeddings.

Preffered hyper parameter values:

| |
|---|
| Dimension = 300 |
| window = 8 |

NOTES ⟹ skip gram model gives better results most no of times.
However in gensim library, CBOW is default for Word2Vec.

Sg = 1 : Skip-gram
otherwise : CBOW.

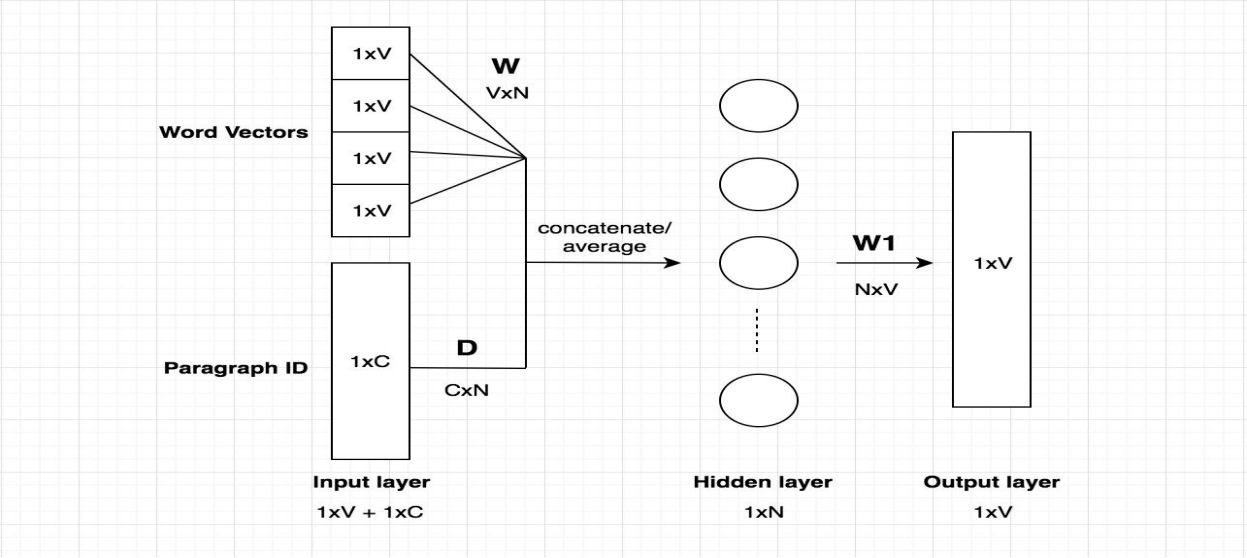
2017

**Advantages of Skip-Gram Model**

1. **Skip-gram model can capture two semantics for a single word. i.e it will have two vector representations of Apple. One for the company and other for the fruit.**
2. **Skip-gram with negative sub-sampling outperforms every other method generally.**

Doc2Vec:

Doc2vec model is based on Word2Vec, with only adding another vector (paragraph ID) to the input. The architecture of Doc2Vec model is shown below:




**The above diagram is based on the CBOW model, but instead of using just nearby words to predict the word, we also added another feature vector, which is document-unique. So when training the word vectors W, the**

**document vector D is trained as well, and in the end of training, it holds a numeric representation of the document.**

**The inputs consist of word vectors and document Id vectors. The word vector is a one-hot vector with a dimension `1xV`. The document Id vector has a dimension of `1xC`, where C is the number of total documents. The dimension of the weight matrix W of the hidden layer is `VxN`. The dimension of the weight matrix D of the hidden layer is `CxN`.**

**The above model is called distributed Memory version of Paragraph Vector (PV-DM). Another Doc2Vec algorithm which is based on Skip-Gram is called Distributed Bag of Words version of Paragraph Vector (PV-DBOW).**

## FastText:

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

One major draw-back for word-embedding techniques like word2vec and glove was its inability to deal with out of corpus words. These embedding techniques treat word as the minimal entity and try to learn their respective embedding vector. Hence in case there is a word that does not appear in the corpus word2vec or glove fails to get their vectorized representation. However fasttext follows the same skipgram and cbow model like word2vec.

It treats each word as composed of n-grams. That is let us say value of n is 3 for the word **‘India’** we have ‘<in’, ‘ind’, ‘ndi’, ‘di>’ as the n-gram representation. And for the word ‘India’ we can infer the whole vector as sum of the vector representation all the character n-grams.(*Here it is assumed that the hyperparameter [minn] and [maxn] value is 3,where ‘minn’ and ‘maxn’ are the smallest and largest ngram respectively)*. The symbols ‘<’ and ‘>’ are special symbols and are appended to show the start and end of the token.(*P.S <her> and ‘her’ are not the same.)*

**ONE CAN TAKE ANY WORD AND APPEND UN OR LY AT END AND A WORD THAT IS NOT IN THE TRAINING SET CAN BE FORMED.**

Fasttext can generate embedding for the words that does not appear in the training corpus. This can be done by adding the character n-gram of all the n-gram representations. For example,let’s say there is a word ‘*commonly*’ in the testing dataset,but doesn’t have any representation in the training set. But training set has vector representation of all its n-grams. So we can just average the vectorized representation of all its constituent n-grams.word. On the otherhand for a random word ‘*fgghoio*’ we can get the representation of by average all ngram characters(i.e ‘f’+ ‘g’ + ‘g’ + ‘h’ + ‘o’ + ‘i’ + ‘o’ *here we have to keep the hyperparameter minn as 1*).

**DrawBack:**

One of the major drawback of this model is high memory requirement. Since this model creates word-embedding from its characters and not from words. We can control the number of character embeddings by using the maximum and minimum ngrams. It has been seen that for a corpus size of 50 million unique words the requires ram size can be as much as 256 GB RAM. Hence in order to get rid off this problem we can make use of a hyper-parameter called min word count, which can be increased to ignore words below a certain threshold.

Fasttext can be used both for classification and word-embedding creation.

**SOME USEFUL LINKS:**

https://towardsdatascience.com/topic-modelling-with-plsa-728b92043f41

https://towardsdatascience.com/supercharging-word-vectors-be80ee5513d

https://medium.com/data-science-group-iitr/word-embedding-2d05d270b285

https://machinelearningmastery.com/what-are-word-embeddings/

https://www.analyticsvidhya.com/blog/2017/05/neural-network-from-scratch-in-python-and-r/

http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/

https://www.quora.com/How-is-GloVe-different-from-word2vec

https://medium.com/@jonathan_hui/nlp-word-embedding-glove-5e7f523999f6

https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1194/slides/cs224n-2019-lecture02-wordvecs2.pdf

https://shuzhanfan.github.io/2018/08/understanding-word2vec-and-doc2vec/

https://www.youtube.com/watch?v=ASn7ExxLZws
https://www.youtube.com/watch?v=ERibwqs9p38

https://arxiv.org/pdf/1607.01759.pdf

https://arxiv.org/pdf/1607.04606.pdf
https://fasttext.cc/docs/en/unsupervised-tutorial.html
https://stackabuse.com/python-for-nlp-working-with-facebook-fasttext-library/