Assignment 5
Title : Implement the Continuous Bag of Words (CBOW) Model.

Aim: Implement the Continuous Bag of Words (CBOW) Model. Stages can be:

a. Data preparation

b. Generate training data

c. Train model

d. Output

Theory :

In Natural Language Processing, we want computers to understand the text as we humans do. However, for this to happen, we need them to translate the words to a language computers can work with and understand.

Word embedding is a numerical representation of words, such as how colors can be represented using the RGB system.

A common representation is one-hot encoding. This method encodes each word with a different vector. The size of the vectors equals the number of words. Thus, if there are 1.0000 words, the vectors have a size of 1X10000. All values in the vectors are zeros except for a value 1, which differentiates each word representation.

Let's use the following sentence, *the pink horse is eating*. In this example, there are words; therefore, the size of the vectors is 1X5. Let's see how it works:

$$\text{the pink horse is eating}$$

$$\text{the} \rightarrow \begin{bmatrix}1\\0\\0\\0\\0\end{bmatrix} \quad \text{pink} \rightarrow \begin{bmatrix}0\\1\\0\\0\\0\end{bmatrix} \quad \text{horse} \rightarrow \begin{bmatrix}0\\0\\1\\0\\0\end{bmatrix} \quad \text{is} \rightarrow \begin{bmatrix}0\\0\\0\\1\\0\end{bmatrix} \quad \text{eating} \rightarrow \begin{bmatrix}0\\0\\0\\0\\1\end{bmatrix}$$

Each word gets assigned a different vector; however, this representation involves different issues. First of all, if the vocabulary is too big, the size of the vectors will be huge. This would lead to the curse of dimensionality when using a model with this encoding. Also, if we add or remove words to the vocabulary, the representation of all words will change.

However, the most important problem of using one-hot encoding is that it doesn't encapsulate meaning. It is just a numeration system to distinguish words. If we want computers to read text as humans do, we need embeddings that capture semantic and syntactic information. The values in the vectors must somehow quantify the meaning of the words they represent.

Word2Vec is a common technique used in Natural Language Processing. In it, similar words have similar word embeddings; this means that they are close to each other in terms of cosine distance. There

are two main algorithms to obtain a Word2Vec implementation: Continous Bag of Words and Skip-Gram. These algorithms use neural network models in order to obtain the word vectors.

These models work using context. This means that the embedding is learned by looking at nearby words; if a group of words is always found close to the same words, they will end up having similar embeddings. Thus, countries will be closely related, so will animals, and so on.

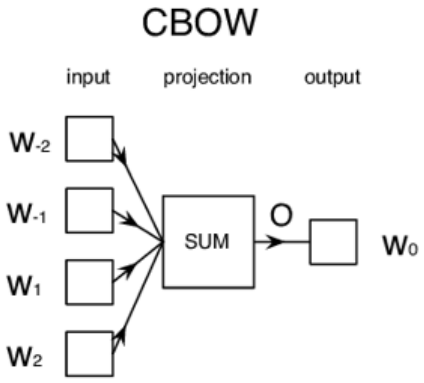
To label how words are close to each other, we first set a window-size. The window-size determines which nearby words we pick. For example, given a window-size of 2, for every word, we'll pick the 2 words behind it and the 2 words after it:

Sentence	Word pairs
the pink horse is eating	(**the**, pink), (**the**, horse)
the **pink** horse is eating	(**pink**, the), (**pink**, horse), (**pink**, is)
the pink **horse** is eating	(**horse**, the), (**horse**, pink), (**horse**, is), (**horse**, eating)
the pink horse **is** eating	(**is**, pink), (**is**, horse), (**is**, eating)
the pink horse is **eating**	(**eating**, horse), (**eating**, is)

In the table above, we can see the word pairs constructed with this method. The highlighted word is the one we are finding pairs for. We don't care about how far the words in the window are. We don't differentiate between words that are 1 word away or more, as long as they're inside the window.

In CBOW From the context words, we want our model to predict the main word:

In the CBOW model, the distributed representations of context (or surrounding words) are combined to predict the word in the middle. While in the Skip-gram model, the distributed representation of the input word is used to predict the context.



The CBOW model architecture is as shown above. The model tries to predict the target word by trying to understand the context of the surrounding words. Consider the same sentence as above, 'It is a pleasant day'.The model converts this sentence into word pairs in the form (contextword, targetword). The user will have to set the window size. If the window for the context word is 2 then the word pairs would look like this: ([it, a], is), ([is, pleasant], a),([a, day], pleasant). With these word pairs, the model tries to predict the target word considered the context words.

If we have 4 context words used for predicting one target word the input layer will be in the form of four 1XW input vectors. These input vectors will be passed to the hidden layer where it is multiplied by a WXN matrix. Finally, the 1XN output from the hidden layer enters the sum layer where an element-wise summation is performed on the vectors before a final activation is performed and the output is obtained.

Advantages of CBOW:

1. Generally, it is supposed to perform superior to deterministic methods due to its probabilistic nature.

2. It does not need to have huge RAM requirements. So, it is low on memory.

Disadvantages of CBOW:

1. CBOW takes the average of the context of a word. **For Example,** consider the word apple that 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. If we want to train a CBOW model from scratch, then it can take forever if we not properly optimized it.

Continuous Bag of Words (CBOW) single-word model Implementation:

In this section we will be implementing the CBOW for single-word architecture of Word2Vec. The content is broken down into the following steps:

Data Preparation: Defining corpus by tokenizing text.

Generate Training Data: Build vocabulary of words, one-hot encoding for words, word index.

Train Model: Pass one hot encoded words through **forward pass**, calculate error rate by computing loss, and adjust weights using **back propagation**.

Output: By using trained model calculate word vector and find similar words.

1. Data Preparation:

Let's say we have a text like below:

"i like natural language processing"

To make it simple I have chosen a sentence without capitalization and punctuation. Also I will not remove any stop words ("and", "the" etc.) but for real world implementation you should do lots of cleaning task like stop word removal, replacing digits, remove punctuation etc.

After pre-processing we will convert the text to list of tokenized word.

["i", "like", "natural", "language", "processing"]

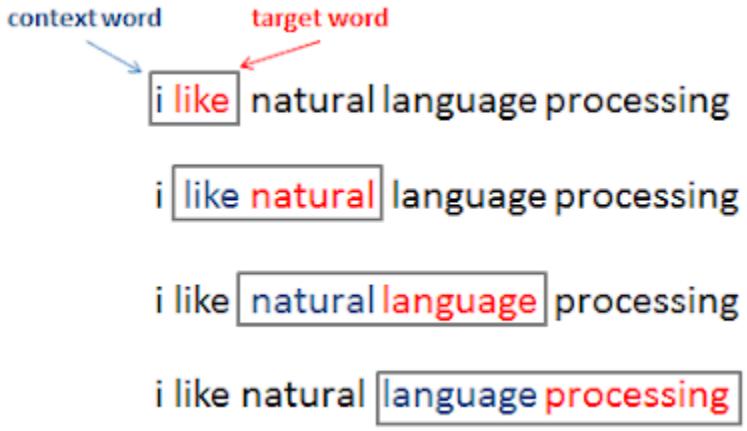
2. Generate training data:

Unique vocabulary: Find unique vocabulary list. As we don't have any duplicate word in our example text, so unique vocabulary will be:

["i", "like", "natural", "language", "processing"]

Now to prepare training data for single word CBOW model, we define "target word" as the word which follows a given word in the text (which will be our "context word"). That means we will be predicting next word for a given word.

Now let's construct our training examples, scanning through the text with a window will prepare a context word and a target word, like so:



For example, for context word "**i**" the target word will be "**like**". For our example text full training data will looks like:

Training Example	Context Word	Target Word
#1	i	like
#2	like	natural
#3	natural	language
#4	language	processing

One-hot encoding: We need to convert text to one-hot encoding as algorithm can only understand numeric values.

For example encoded value of the word "**i**", which appears first in the vocabulary, will be as the vector [1, 0, 0, 0, 0]. The word "like", which appears second in the vocabulary, will be encoded as the vector [0, 1, 0, 0, 0]

	i	like	natural	language	processing
i	1	0	0	0	0
like	0	1	0	0	0
natural	0	0	1	0	0
language	0	0	0	1	0
processing	0	0	0	0	1

So let's see overall set of context-target words in one hot encoded form:

Training Example	Encoded Context Word	Encoded Target Word
#1	[1,0,0,0,0]	[0,1,0,0,0]
#2	[0,1,0,0,0]	[0,0,1,0,0]
#3	[0,0,1,0,0]	[0,0,0,1,0]
#4	[0,0,0,1,0]	[0,0,0,0,1]

So as you can see above table is our final training data, where encoded target word is **Y variable**for our model and encoded context word is **X variable** for our model.

Now we will move on to train our model.

3. Training Model:

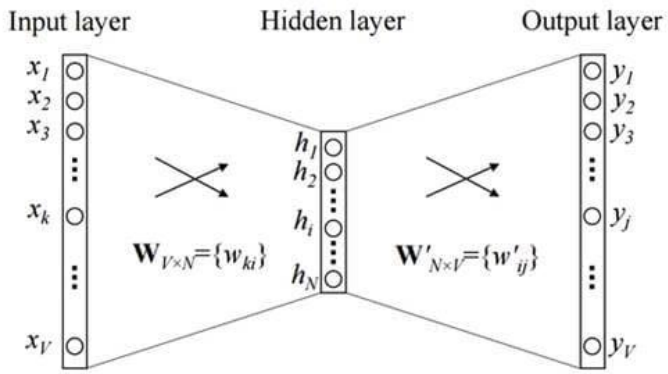


Figure 1: A simple CBOW model with only one word in the context

So far, so good right? Now we need to pass this data into the basic neural network with one hidden layer and train it. Only one thing to note is that the desire vector dimension of any word will be the number of hidden nodes.

For this tutorial and demo purpose my desired vector dimension is 3. For example:

"i" => [0.001, 0.896, 0.763] so number of hidden layer node will be 3.

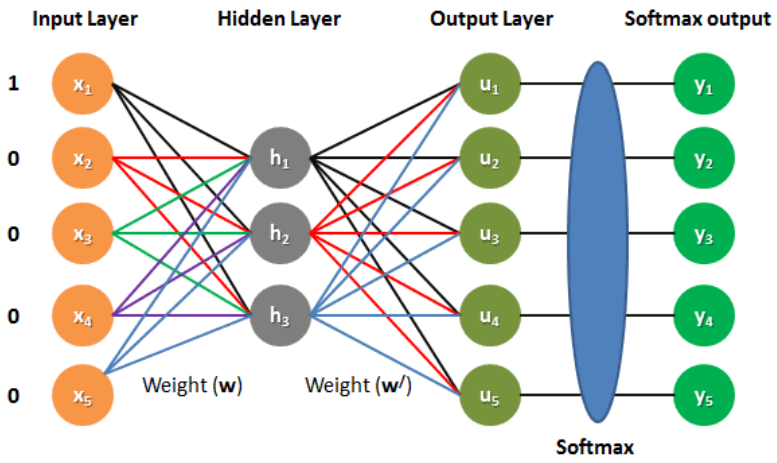
Dimension (n): It is dimension of word embedding you can treat embedding as number of features or entity like organization, name, gender etc. It can be 10, 20, 100 etc. Increasing number of embedding layer will explain a word or token more deeply. Just for an example Google pre-trained word2vec have dimension of 300.

Now as you know a basic neural network training is divided into some steps:

1. Create model Architecture
2. Forward Propagation
3. Error Calculation
4. Weight tuning using backward pass

Before going into forward propagation we need to understand model architecture in vectorized form.

3.1 Model Architecture:

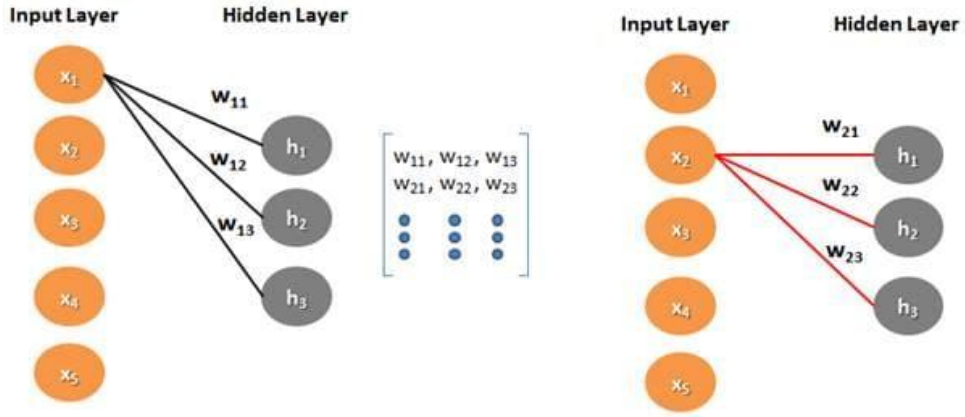


First training data point: The context word is "**i**" and the target word is "**like**".

Let's look back to our example text "I like natural language processing". Suppose we are training the model against the first training data point where the context word is "**i**" and the target word is "like".

Ideally, the values of the weights should be such that when the input x=(1,0,0,0) – which is for the word "**i**" is given to the model –it will produce output which should close to y=(0,1,0,0) – which is the word "like".

So now let's create weight matrix for input to hidden layer (**w**).



Creating weight matrix for input to hidden layer

Above picture shows how I have created weighted matrix for first two input nodes. So in the similar manner if we are creating weight matrix for all input nodes it should looks like below, whose dimension will be [3 X 5], in different way it's

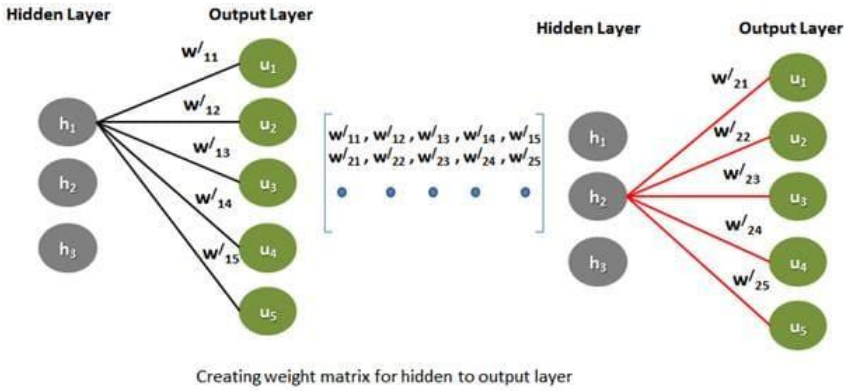
[N x V]

Where:

N: *Number of embedding layer/ hidden unit*

V: *Number of unique vocabulary size*

Now let's create weight matrix for hidden to output layer ($\mathbf{w}^{/}$).



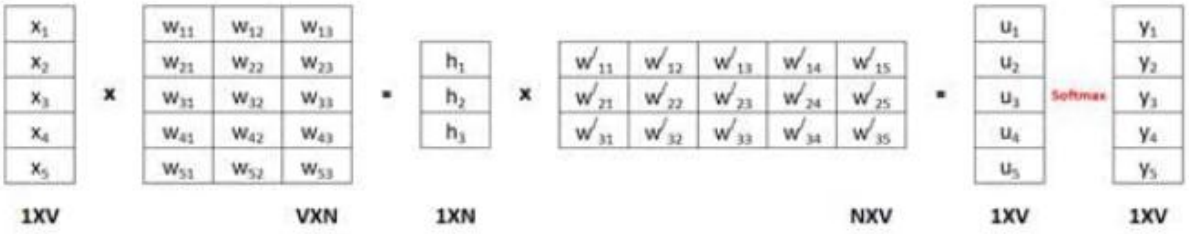
Creating weight matrix for hidden to output layer

So in the similar manner (shown in the above picture) if we are creating weight matrix for all hidden nodes it should looks like below, whose dimension will be [5 X 3], in different way it's **[V X N]**

$$w = \begin{bmatrix} w'_{11} & w'_{12} & w'_{13} & w'_{14} & w'_{15} \\ w'_{21} & w'_{22} & w'_{23} & w'_{24} & w'_{25} \\ w'_{31} & w'_{32} & w'_{33} & w'_{34} & w'_{35} \end{bmatrix}$$

So final vectorized form of CBOW model should look like below:

CBOW Vectorized form:



CBOW vectorixed form

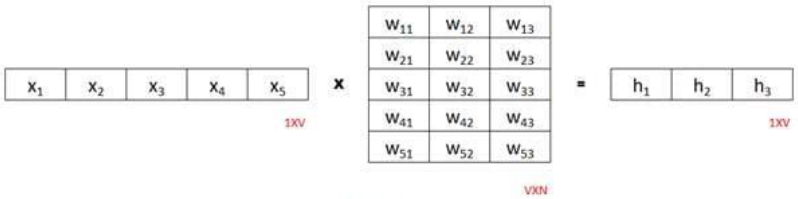
Where **V** is number of unique vocabulary and **N** is number of embedding layers (number of hidden units)

Now we can start forward propagation.

3.2 Forward Propagation:

Please refer to picture of CBOW vectorized form, if you have any confusion in below calculations.

Calculate hidden layer matrix (h):



Hiddel Layer matrix calculation

So now:

$$h_1 = w_{11}x_1 + w_{21}x_2 + w_{31}x_3 + w_{41}x_4 + w_{51}x_5$$
$$h_2 = w_{12}x_1 + w_{22}x_2 + w_{32}x_3 + w_{42}x_4 + w_{52}x_5$$
$$h_3 = w_{13}x_1 + w_{23}x_2 + w_{33}x_3 + w_{43}x_4 + w_{53}x_5$$

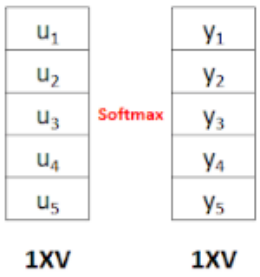
Calculate output layer matrix (u):

Output Layer matrix calculation

So now:

$$u_1 = w'_{11}h_1 + w'_{21}h_2 + w'_{31}h_3$$
$$u_2 = w'_{12}h_1 + w'_{22}h_2 + w'_{32}h_3$$
$$u_3 = w'_{13}h_1 + w'_{23}h_2 + w'_{33}h_3$$
$$u_4 = w'_{14}h_1 + w'_{24}h_2 + w'_{34}h_3$$
$$u_5 = w'_{15}h_1 + w'_{25}h_2 + w'_{35}h_3$$

Calculate final Softmax output (y):



So:

y_1 = Softmax (u_1)

y_2 = Softmax (u_2)

y_3 = Softmax (u_3)

y_4 = Softmax (u_4)

y_5 = Softmax (u_5)

Now as you know softmax calculate probability for every possible class. Softmax function uses exponential as we want to get output from softmax in a range between 0 to 1.

Let's have a look for only one (first) output from softmax function:

$$y_1 = \frac{e^{u_1}}{(e^{u_1} + e^{u_2} + e^{u_3} + e^{u_4} + e^{u_5})}$$

So now if we want to generalize above equation (as the above equation for first output from softmax) we can write:

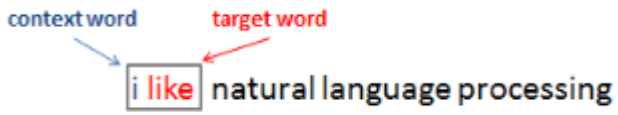
$$y_j = \frac{e^j}{\sum_{j=1}^{V} e^j}$$

3.3 Error Calculation:

As we have done with forward propagation, now we need to calculate error to know how accurately our model is performing and update weights (**w and w/**) accordingly.

As you know to calculate error we must have an actual value. That actual value needs to compare with predicted value.

For single word model next word is the target word for previous word.



Here we will be using log loss function to calculate error. First let's have a look at generalized equation to minimize error/ log loss:

[E=-log(w_t|w_c)]

Where

$\mathbf{w_t}$ = Target word

$\mathbf{w_c}$ = Context word

So now let's calculate error / loss **for first iteration** only. For the first iteration "**like**" will be the target word and its position is 2.

So,

$$E(y_2) = -log(w_{y_2}|w_{x_1})$$

$$= -log\frac{e^{u_2}}{e^{u_1} + e^{u_2} + e^{u_3} + e^{u_4} + e^{u_5}}$$

$$= -log(e^{u_2}) + log(e^{u_1} + e^{u_2} + e^{u_3} + e^{u_4} + e^{u_5})$$

$$= -u_2 + log(e^{u_1} + e^{u_2} + e^{u_3} + e^{u_4} + e^{u_5})$$

So now if we want to generalize above equation we can write like this:

$$E = -u_{j^*} + log\sum_{j=1}^{V} e^{u_j}$$

Where j* is the index of the actual word (target word) in the output layer. For first iteration index of target word is 2. So for first iteration j* will be 2 as position of word "**like**" is 2.

3.5 Back Propagation:

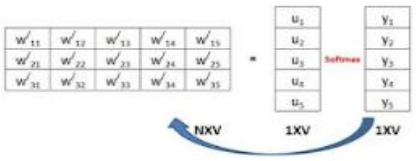
As we have done with forward propagation and error calculation now we need to tune weight matrices through back propagation.

Back propagation is required to update weight matrices (**w and w/**). To update weight we need to calculate derivative of loss with respect to each weight and subtract those derivatives with their corresponding weight. It's called **gradient descent** technique to tune weight.

Let's take a look for only one example to update second weight (**w/**).

In this back propagation phase we will update weight of all neurons from hidden layer to output (w'_{11}, w'_{12}, w'_{13} .... w'_{15})

Step1: Gradient of E with respect to w'_{11}:



$$\frac{dE(y_1)}{dw'_{11}} = \frac{dE(y_1)}{du_1} . \frac{du_1}{dw'_{11}}$$

Now as we know

$$E(y_1) = -u_1 + log(e^{u_1} + e^{u_2} + e^{u_3} + e^{u_4} + e^{u_5})$$

$$So, \frac{dE(y_1)}{du_1} = -\frac{du_1}{du_1} + \frac{d[log(e^{u_1} + e^{u_2} + e^{u_3} + e^{u_4} + e^{u_5})])}{du_1}$$

$$= -1 + \frac{d[log(e^{u_1} + e^{u_2} + e^{u_3} + e^{u_4} + e^{u_5})]}{d(e^{u_1} + e^{u_2} + e^{u_3} + e^{u_4} + e^{u_5})} . \frac{d(e^{u_1} + e^{u_2} + e^{u_3} + e^{u_4} + e^{u_5})}{du_1}$$

Derivative of log function with chain rule.

$$= -1 + \frac{1}{e^{u_1} + e^{u_2} + e^{u_3} + e^{u_4} + e^{u_5}} * u_1$$

$$= -1 + \frac{u_1}{e^{u_1} + e^{u_2} + e^{u_3} + e^{u_4} + e^{u_5}}$$

$$= -1 + y_1$$

Taking derivative of **E** with respect to $\mathbf{u_j}$:

$$\frac{dE}{du_j} = -\frac{d(u_{j*})}{du_j} + \frac{d(log\sum_{j=1}^{V} e^{u_j})}{du_j}$$
$$= (-t_j + y_j)$$
$$= (y_j - t_j)$$
$$= e_j$$

Note: t_j=1 if $t_j = t_{j*}$ else $t_j = 0$

That means t_j is the actual output and y_j is the predicted output. So e_j is the error.

So for first iteration,

$$\frac{dE(y_1)}{du_1} = e_1$$

$$And,\ \frac{du_1}{dw'_{11}} = \frac{d(w'_{11}h_1 + w'_{21}h_2 + w'_{31}h_3)}{dw'_{11}} = h_1$$

Now finally coming back to main derivative which we were trying to solve:

$$\frac{dE(y_1)}{dw'_{11}} = \frac{dE(y_1)}{du_1}.\frac{du_1}{dw'_{11}} = e_1h_1$$

So generalized form will looks like below:

$$\frac{dE}{dw'} == e * h$$

Step2: Updating the weight of w'_{11}:

$$new(w'_{11}) = w'_{11} - \frac{dE(y_1)}{w'_{11}} = (w'_{11} - e_1h_1)$$

Note: To keep it simple I am not considering any learning rate here.

Similarly we can update weight of w'_{12}, w'_{13} .... w'_{35}.

Now let's update 1st weight which (w).

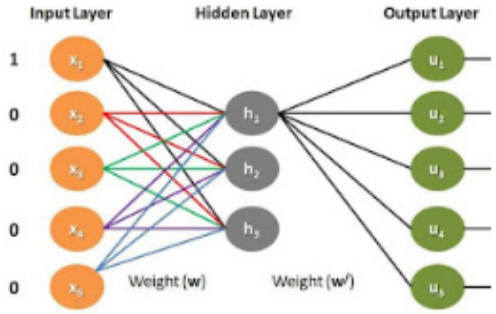
Step1: Gradient of E with respect to w_{11}:

In this back propagation phase we will update weight of all neurons from input layer to hidden layer (w_{11}, w_{12}, ...w_{53})

Again I will take you through only first weigh (w_{11})

$$\frac{dE}{dw_{11}} = \frac{dE}{dh_1}, \frac{dh_1}{dw_{11}}$$

As to change E with respect to h_1, $u_1$to u_5 all are involved.



So,

$$\frac{dE}{dh_1} = (\frac{dE}{du_1}, \frac{du_1}{dh_1}) + (\frac{dE}{du_2}, \frac{du_2}{dh_1}) + (\frac{dE}{du_3}, \frac{du_3}{dh_1}) + (\frac{dE}{du_4}, \frac{du_4}{dh_1}) + (\frac{dE}{du_5}, \frac{du_5}{dh_1})$$

$$= ew_{11}^{'} + ew_{12}^{'} + ew_{13}^{'} + ew_{14}^{'} + ew_{15}^{'}$$

As for u_1 and h_1,

$$\frac{du_1}{dh_1} = \frac{d(w'_{11}h_1 + w'_{21}h_2 + w'_{31}h_3)}{dh_1} = w'_{11}$$

$$As,\ h_2\ and\ h_3\ are\ constant\ with\ respect\ to\ h_1$$

$$Similarly\ we\ can\ calculate\ :\ \frac{du_2}{dh_1}, \frac{du_3}{dh_1}, \frac{du_4}{dh_1}, \frac{du_5}{dh_1}]$$

$$And,\ \frac{dh_1}{dw_{11}} = \frac{d(w_{11}x_1 + w_{21}x_2 + w_{31}x_3 + w_{41}x_4 + w_{51}x_5)}{dw_{11}}$$

Now finally:

$$\frac{dE}{dw_{11}} = \frac{dE}{dh_1}, \frac{dh_1}{dw_{11}}$$
$$= (ew'_{11} + ew'_{12} + ew'_{13} + ew'_{14} + ew'_{15}) * x$$

Step2: Updating the weight of w_{11}:

$$new(w_{11}) = w_{11} - \frac{dE}{dw_{11}}$$
$$= w_{11} - (ew'_{11} + ew'_{12} + ew'_{13} + ew'_{14} + ew'_{15}) * x$$

Note: Again to keep it simple I am not considering any learning rate here.

Similarly we can update weight of w_{12}, w_{13} … W_{53}.

Final CBOW Word2Vec:

After doing above training process for many iteration final trained model comes with tuned and proper weights. At the end of entire training process we will consider first weight matrix (w) as our vector for our given sentence.

For example:

Our sentence or document was"i like natural language processing"

And let's say first weight of our trained model is:

$$w = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \\ w_{41} & w_{42} & w_{43} \\ w_{51} & w_{52} & w_{53} \end{bmatrix}$$

So word2vce of word "i" will be [w_{11}, w_{12}, w_{13}]

Similarly word2vec for all word will be as follows:

"i"	w11	w12	w13
"like"	w21	w22	w23
"natural"	w31	w32	w33
"language"	w41	w42	w43
"processing"	w51	w52	w53

Q&A

1)What is NLP ?

2) What is Word embedding related to NLP ?

3) Explain Word2Vec techniques.

4) Enlist applications of Word embedding in NLP.

5) Explain CBOW architecture.

6) What will be input to CBOW model and Output to CBW model.

7) What is Tokenizer .

8) Explain window size parameter in detail for CBOW model.

9) Explain Embedding and Lmbda layer from keras

10) What is yield()

Steps/ Algorithm

1. Dataset link and libraries :

Create any English 5 to 10 sententece paragraph as input

Import following data from keras :

keras.models import Sequential

keras.layers import Dense, Embedding, Lambda

keras.utils import np_utils

keras.preprocessing import sequence

keras.preprocessing.text import Tokenizer

Import Gensim for NLP operations : requirements :

Gensim runs on Linux, Windows and Mac OS X, and should run on any other platform that supports Python 3.6+ and NumPy. Gensim depends on the following software: Python, tested with versions 3.6, 3.7 and 3.8. NumPy for number crunching.

Ref: https://analyticsindiamag.com/the-continuous-bag-of-words-cbow-model-in-nlp-hands-onimplementation-with-codes/

a) Import following libraries gemsim and numpy set i.e. text file created . It should be preprocessed.

b) Tokenize the every word from the paragraph . You can call in built tokenizer present in Gensim

c) Fit the data to tokenizer

d) Find total no of words and total no of sentences.

e) Generate the pairs of Context words and target words :

e.g. cbow_model(data, window_size, total_vocab):

```
total_length = window_size*2
for text in data:
text_len = len(text)
for idx, word in enumerate(text):
context_word = []
target = []
begin = idx - window_size
end = idx + window_size + 1
context_word.append([text[i] for i in range(begin, end) if 0 <= i < text_len and i
!= idx])
target.append(word)
contextual = sequence.pad_sequences(context_word, total_length=total_length)
final_target = np_utils.to_categorical(target, total_vocab)
yield(contextual, final_target)
```

f) Create Neural Network model with following parameters . Model type : sequential Layers : Dense , Lambda , embedding. Compile Options : (loss='categorical_crossentropy', optimizer='adam')

g) Create vector file of some word for testing

e.g.:dimensions=100

```
vect_file = open('/content/gdrive/My Drive/vectors.txt' ,'w')
vect_file.write('{} {}\n'.format(total_vocab,dimensions)
```

h) Assign weights to your trained model

```
e.g. weights = model.get_weights()[0]
for text, i in vectorize.word_index.items():
final_vec = ' '.join(map(str, list(weights[i, :])))
vect_file.write('{} {}\n'.format(text, final_vec)
Close()
```

i) Use the vectors created in Gemsim :

```
e.g. cbow_output =
gensim.models.KeyedVectors.load_word2vec_format('/content/gdrive/My
Drive/vectors.txt', binary=False)
```

j) choose the word to get similar type of words:

```
cbow_output.most_similar(positive=['Your word'])
```

Sample Code with comments : Attach Printout with Output .

Conclusion: Explain how Neural network is useful for CBOW text analysis.