GeeksforGeeks
A computer science portal for geeks

Implement your own … -gram) model in Python

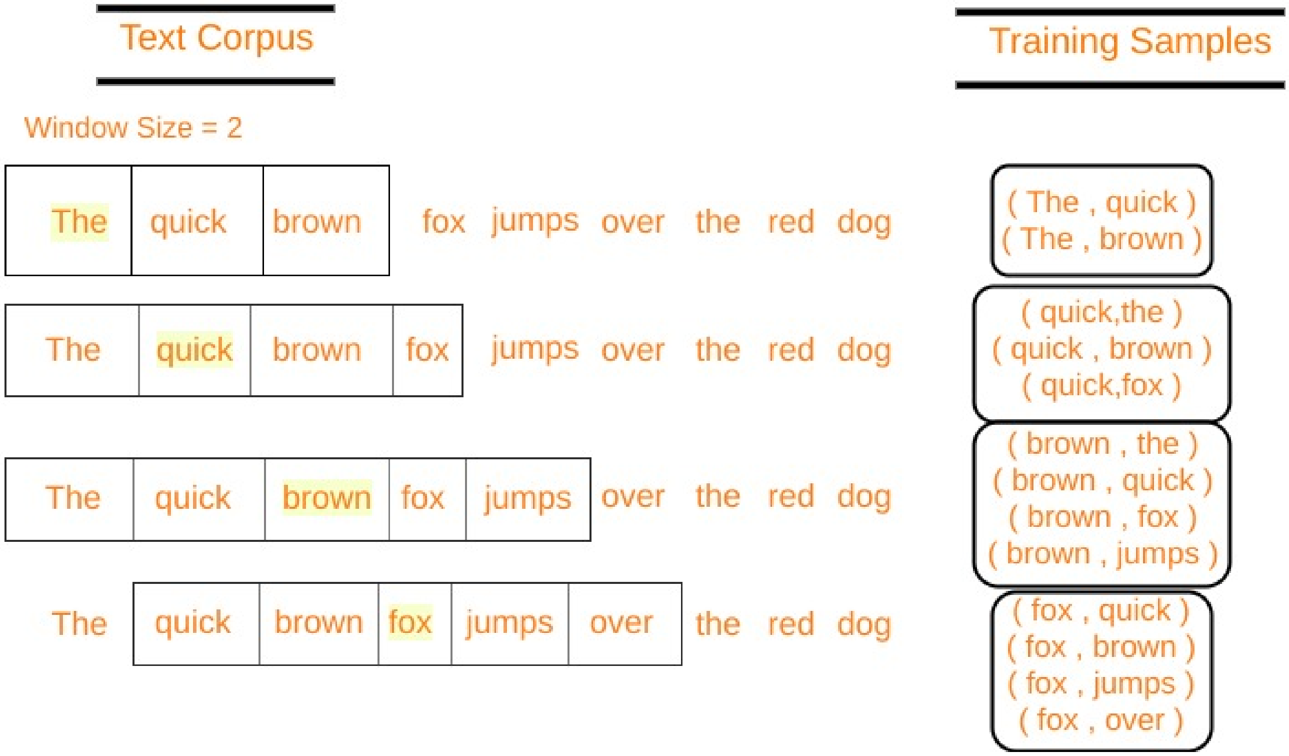
Prerequisite: Introduction to word2vec

Natural language processing (NLP) is a subfield of computer science and artificial intelligence concerned with the interactions between computers and human (natural) languages.
In NLP techniques, we map the words and phrases (from vocabulary or corpus) to vectors of numbers to make the processing easier. These types of **language modeling** techniques are called **word embeddings**.

In 2013, Google announched **word2vec**, a group of related models that are used to produce word embeddings.

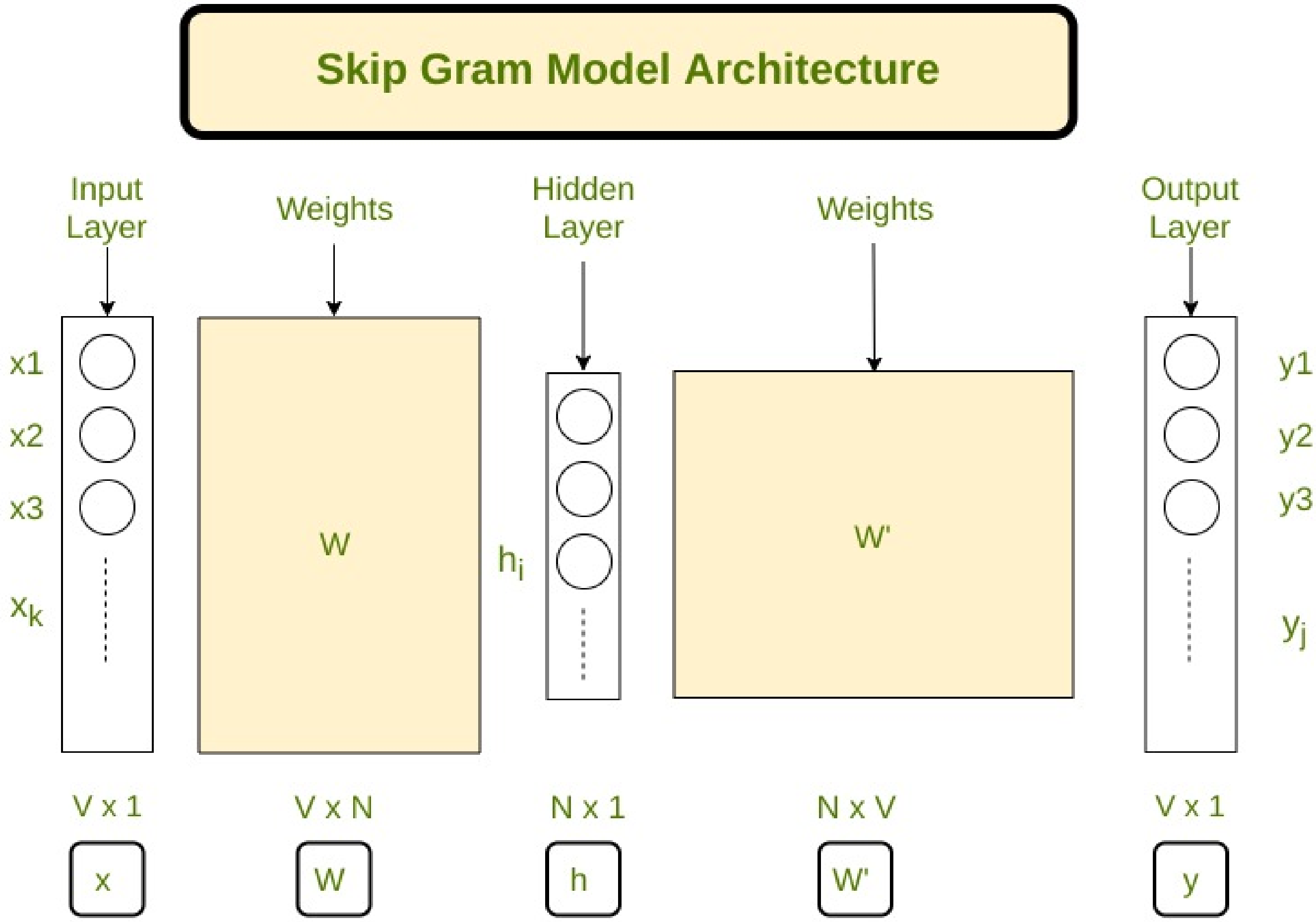
Let's implement our own skip-gram model (in Python) by deriving the backpropagation equations of our neural network.

In **skip gram** architecture of word2vec, the input is the **center word** and the predictions are the context words. Consider an array of words W, if W(i) is the input (center word), then W(i-2), W(i-1), W(i+1), and W(i+2) are the context words, if the *sliding window size* is 2.



Let's define some variables :

V Number of unique words in our corpus of text (**V**ocabulary)
x Input layer (One hot encoding of our input word).
N Number of neurons in the hidden layer of neural network
W Weights between input layer and hidden layer
W' Weights between hidden layer and output layer
y A softmax output layer having probabilities of every word in our vocabulary



Skip gram architecture

Our neural network architecture is defined, now let's do some math to derive the equations needed for gradient descent.

Forward Propagation:

Multiplying one hot encoding of centre word (denoted by **x**) with the first weight matrix **W** to get hidden layer matrix **h** (of size N x 1).

$$h = W^T.x$$

(Vx1)   (NxV) (Vx1)

Now we multiply the hidden layer vector **h** with second weight matrix **W'** to get a new matrix **u**

$$u = W'^T.h$$

(Vx1)   (VxN) (Nx1)

Note that we have to apply a softmax> to *layer u* to get our *output layer y*.

Let $\mathbf{u_j}$ be j^{th} neuron of layer **u**
Let $\mathbf{w_j}$ be the j^{th} word in our vocabulary where j is any index
Let $\mathbf{V_{w_j}}$ be the j^{th} column of matrix **W'**(column corresponding to a **word** $\mathbf{w_j}$)

$$u_j = V_{w_{ij}}^T . h$$
(1×1)   (1xN) (Nx1)

y = softmax(u)

y_j = softmax(u_j)

y_j denotes the probability that w_j is a context word

$$P(w_j|w_i) = y_j = \frac{e^{u_j}}{\sum_{j'=1}^{v} e^{u_{j'}}}$$

P(w_j|w_i) is the probability that w_j is a context word, given w_i is the input word.

Thus, our goal is to maximise **P(w_{j*} | w_i)**, where j* represents the indices of context words

Clearly we want to maximise

$$\prod_{c=1}^{C} \frac{e^{u_{j_c*}}}{\sum_{j'=1}^{v} e^{u_{j'}}}$$

where **$j*_c$** are the vocabulary indexes of context words . Context words range from **c = 1, 2, 3..C**

Let's take a **negative log likelihood** of this function to get our **loss function**, which we want to **minimise**

$$E = -\log\left\{\prod_{c=1}^{C} \frac{e^{u_{j_c*}}}{\sum_{j'=1}^{v} e^{u_{j'}}}\right\}, \ E\ being\ our\ loss\ function$$

Let *t* be actual output vector from our training data, for a particular centre word. It will have 1's at the positions of context words and 0's at all other places. t_{j*_c} are the 1's of the context words.

We can multiply u_{j_c*} with t_{j_c*}

$$E = -log(\prod_{c=1}^{C} e^{u_{j_c*}}) + log(\sum_{j'=1}^{v} e^{u_{j'}})^C$$

Solving this equation we get our loss function as –

$$E = -\sum_{c=1}^{C} u_{j_c*} + C.log(\sum_{j'=1}^{v} e^{u_{j'}})$$

Back Propagation:

The parameters to be adjusted are in the matrices W and W', hence we have to find the partial derivatives of our loss function with respect to W and W' to apply gradient descent algorithm.

We have to find $\frac{\partial E}{\partial W'}\ and\ \frac{\partial E}{\partial W}$

$$\frac{\partial E}{\partial w'_{ij}} = \frac{\partial E}{\partial u_j} . \frac{\partial u_j}{\partial w'_{ij}}$$

$$\frac{\partial E}{\partial u_j} = -\sum_{c=1}^{C} u_{j_c*} + C.\frac{1}{\sum_{j'=1}^{v} e^{u_{j'}}} . \frac{\partial}{\partial u_j}\sum_{j=1}^{V} e^{u_j}$$

$$\frac{\partial E}{\partial u_j} = -\sum_{c=1}^{C} 1 + \sum_{j=1}^{V} y_j$$

$$\frac{\partial E}{\partial u_j} = y_j - t_j = e_j$$

$$\frac{\partial E}{\partial w'_{ij}} = e_j . \frac{\partial u_j}{\partial w'_{ij}} = e_j . \frac{\partial w'_{ij} * h_i}{\partial w'_{ij}}$$

$$\frac{\partial E}{\partial w'_{ij}} = e_j . h_i$$

Now, Finding $\frac{\partial E}{\partial w_{ij}}$

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial u_j} . \frac{\partial u_j}{\partial w_{ij}}$$

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial u_j} . \frac{\partial u_j}{\partial h_i} . \frac{\partial h_i}{\partial w_{ij}}$$

$$\frac{\partial E}{\partial w_{ij}} = e_j . w'_{ij} . \frac{\partial w_{ij} * x_i}{\partial w_{ij}}$$

$$\frac{\partial E}{\partial w_{ij}} = e_j . w'_{ij} . x_i$$

Below is the implementation :

```
import numpy as np
import string
from nltk.corpus import stopwords

def softmax(x):
    """Compute softmax values for each sets of scores in x."""
    e_x = np.exp(x - np.max(x))
    return e_x / e_x.sum()

class word2vec(object):
    def __init__(self):
        self.N = 10
        self.X_train = []
        self.y_train = []
        self.window_size = 2
        self.alpha = 0.001
        self.words = []
        self.word_index = {}

    def initialize(self,V,data):
        self.V = V
        self.W = np.random.uniform(-0.8, 0.8, (self.V, self.N))
        self.W1 = np.random.uniform(-0.8, 0.8, (self.N, self.V))
```

```
        self.words = data
        for i in range(len(data)):
            self.word_index[data[i]] = i


    def feed_forward(self,X):
        self.h = np.dot(self.W.T,X).reshape(self.N,1)
        self.u = np.dot(self.W1.T,self.h)
        #print(self.u)
        self.y = softmax(self.u)
        return self.y

    def backpropagate(self,x,t):
        e = self.y - np.asarray(t).reshape(self.V,1)
        # e.shape is V x 1
        dLdW1 = np.dot(self.h,e.T)
        X = np.array(x).reshape(self.V,1)
        dLdW = np.dot(X, np.dot(self.W1,e).T)
        self.W1 = self.W1 - self.alpha*dLdW1
        self.W = self.W - self.alpha*dLdW

    def train(self,epochs):
        for x in range(1,epochs):
            self.loss = 0
            for j in range(len(self.X_train)):
                self.feed_forward(self.X_train[j])
                self.backpropagate(self.X_train[j],self.y_train[j])
                C = 0
                for m in range(self.V):
                    if(self.y_train[j][m]):
                        self.loss += -1*self.u[m][0]
                        C += 1
                self.loss += C*np.log(np.sum(np.exp(self.u)))
            print("epoch ",x, " loss = ",self.loss)
            self.alpha *= 1/( (1+self.alpha*x) )

    def predict(self,word,number_of_predictions):
        if word in self.words:
            index = self.word_index[word]
            X = [0 for i in range(self.V)]
            X[index] = 1
            prediction = self.feed_forward(X)
            output = {}
            for i in range(self.V):
                output[prediction[i][0]] = i

            top_context_words = []
            for k in sorted(output,reverse=True):
                top_context_words.append(self.words[output[k]])
                if(len(top_context_words)>=number_of_predictions):
                    break

            return top_context_words
        else:
            print("Word not found in dicitonary")


def preprocessing(corpus):
```

```
    stop_words = set(stopwords.words('english'))
    training_data = []
    sentences = corpus.split(".")
    for i in range(len(sentences)):
        sentences[i] = sentences[i].strip()
        sentence = sentences[i].split()
        x = [word.strip(string.punctuation) for word in sentence
                                     if word not in stop_words]
        x = [word.lower() for word in x]
        training_data.append(x)
    return training_data


def prepare_data_for_training(sentences,w2v):
    data = {}
    for sentence in sentences:
        for word in sentence:
            if word not in data:
                data[word] = 1
            else:
                data[word] += 1
    V = len(data)
    data = sorted(list(data.keys()))
    vocab = {}
    for i in range(len(data)):
        vocab[data[i]] = i

    #for i in range(len(words)):
    for sentence in sentences:
        for i in range(len(sentence)):
            center_word = [0 for x in range(V)]
            center_word[vocab[sentence[i]]] = 1
            context = [0 for x in range(V)]

            for j in range(i-w2v.window_size,i+w2v.window_size):
                if i!=j and j>=0 and j<len(sentence):
                    context[vocab[sentence[j]]] += 1
            w2v.X_train.append(center_word)
            w2v.y_train.append(context)
    w2v.initialize(V,data)

    return w2v.X_train,w2v.y_train



corpus = ""
corpus += "The earth revolves around the sun. The moon revolves around the earth"
epochs = 1000

training_data = preprocessing(corpus)
w2v = word2vec()

prepare_data_for_training(training_data,w2v)
w2v.train(epochs)

print(w2v.predict("around",3))
```

Output:

```
epoch  982  loss =  40.3099405336
epoch  983  loss =  40.3098046028
epoch  984  loss =  40.3096689498
epoch  985  loss =  40.3095335736
epoch  986  loss =  40.3093984735
epoch  987  loss =  40.3092636487
epoch  988  loss =  40.3091290982
epoch  989  loss =  40.3089948212
epoch  990  loss =  40.3088608169
epoch  991  loss =  40.3087270845
epoch  992  loss =  40.3085936232
epoch  993  loss =  40.3084604321
epoch  994  loss =  40.3083275104
epoch  995  loss =  40.3081948572
epoch  996  loss =  40.3080624719
epoch  997  loss =  40.3079303535
epoch  998  loss =  40.3077985013
epoch  999  loss =  40.3076669145
['earth', 'revolves', 'moon']
```

Recommended Posts:

Implement IsNumber() function in Python

Python | Program to implement Jumbled word game

Python | Program to implement simple FLAMES game

Small World Model - Using Python Networkx

Python | Program to implement Rock paper scissor game

Python | ARIMA Model for Time Series Forecasting

Django App Model - Python manage.py makemigrations command

Learning Model Building in Scikit-learn : A Python Machine Learning Library

How to implement Dictionary with Python3?

VGG-16 | CNN model

Implement sigmoid function using Numpy

Bag of words (BoW) model in NLP

5 Machine Learning Projects to Implement as a Beginner

ML | Implement Face recognition using k-NN with scikit-learn

AI Model For Neurodegenerative Diseases