Task - 3 : **Word Embeddings (topic: NLP)**

Submitted by : **Eshaan Agarwal**
Branch : **EEE** | Discord : **Neo_the_one**

Google Colab Link -
https://colab.research.google.com/drive/1qsymw77yOfZy3bRX4CseChxxuXzZbEmV?usp=sharing

OBJECTIVE :

Make a model which will learn the word embeddings of all the words present in this dataset. There is no limitation as to which model you use for developing the embeddings. You have to make the model in Numpy(suggested) or pytorch. Most commonly used are word2vec, glove and fastText. Keep in mind that you have to write down and train the model. Using an api which directly returns the embedding, does not count as completing the question.
After learning the word embeddings (word vectors) visualise the embeddings using manifold learning algorithms.

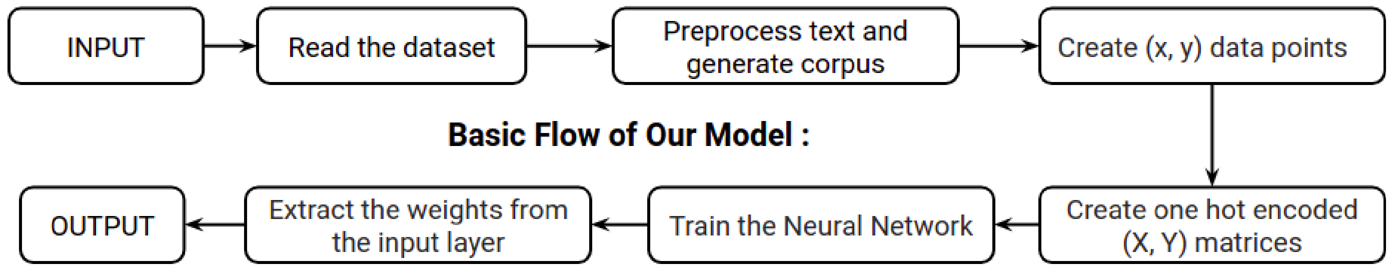
Dataset and Test_Cases Used :

Large Movie Review Dataset(IMDB) from Stanford. The data includes movie reviews along with their associated binary sentiment polarity labels. The core dataset contains 50,000 reviews split evenly into 25k train and 25k test sets. The overall distribution of labels is balanced (25k pos and 25k neg).

MODEL :Word2Vec with One hot encoding vectors(CBOG) using Softmax Activation function ,Cross-Entropy loss function and Adam optimizer

Technologies used : **Keras and TensorFlow and Numpy**

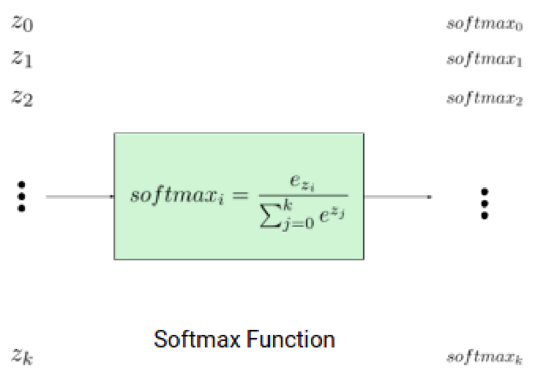
Before we talk about the model to implement that, we need to understand the meaning of **Word Embedding. Word embedding** is the collective name for a set of language modeling and feature learning techniques in natural language processing (NLP) where **words or phrases from the vocabulary are mapped to vectors of real numbers.** One common way of measuring this similarity is to use the cosine of the angle between the vectors. For example, words like “mom” and “dad” should be closer together than the words “mom” and “ketchup” or “dad” and “butter”.
Word embeddings are created using a neural network with one input layer, one hidden layer and one output layer.


INPUT
Read the dataset
Preprocess text and generate corpus
Create (x, y) data points
Create one hot encoded (X, Y) matrices
Train the Neural Network
Extract the weights from the input layer
OUTPUT

Basic Flow of Our Model :

Architecture :

Using functions of the **regex library**, I initially preprocessed the data to remove punctuations and other irrelevant stop words which would decrease the efficiency of our model like **is, the, a, etc.** I **tokenized** it which we can then transform into a matrix form.
After processing,I created a **context dictionary** where all the context words for a given focus word were stored for all the unique words present in the matrix. Since the data set was very large and time consuming, I used **keras data generator** to flush in data to our network in the form of **one-hot vector encodings** of a small batch size. After this I defined and set the parameters of my neural network and created a dictionary for the word embeddings. I used the **Softmax Activation function on output ,Cross-Entropy loss function to compute loss efficiently and update weight and Adam optimizer .**



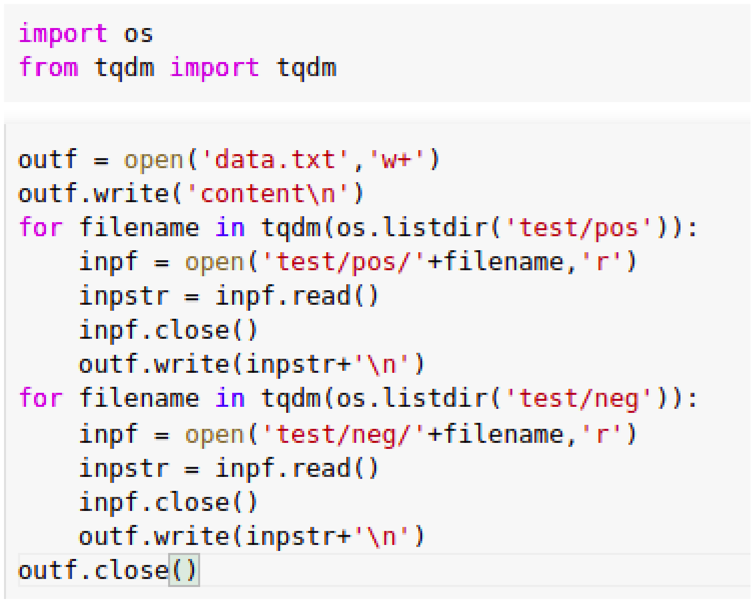
Softmax Function

$$L = -\frac{1}{m} \sum_{j=0}^{m} \sum_{i=0}^{n} y_{ij} \log \hat{y}_{ij}$$

Cross-Entropy Loss Function

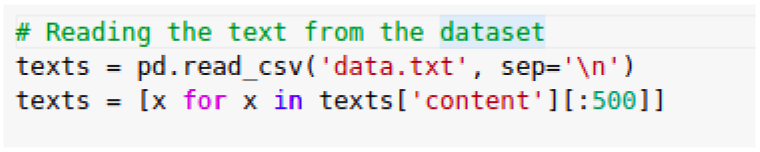
Problem Faced:

1. Merging of the all the .txt files of the dataset into one and preprocessing it
   Solution - With some research and help , I learned about file handling functions present in python.


```
import os
from tqdm import tqdm
```

```
outf = open('data.txt','w+')
outf.write('content\n')
for filename in tqdm(os.listdir('test/pos')):
    inpf = open('test/pos/'+filename,'r')
    inpstr = inpf.read()
    inpf.close()
    outf.write(inpstr+'\n')
for filename in tqdm(os.listdir('test/neg')):
    inpf = open('test/neg/'+filename,'r')
    inpstr = inpf.read()
    inpf.close()
    outf.write(inpstr+'\n')
outf.close()
```

2. I was not able to properly create a list of text as my function was not able to identify where to start.
   Solution : I learned about the delimiter option in pandas and then in my data.txt file initially added a label word “content” .

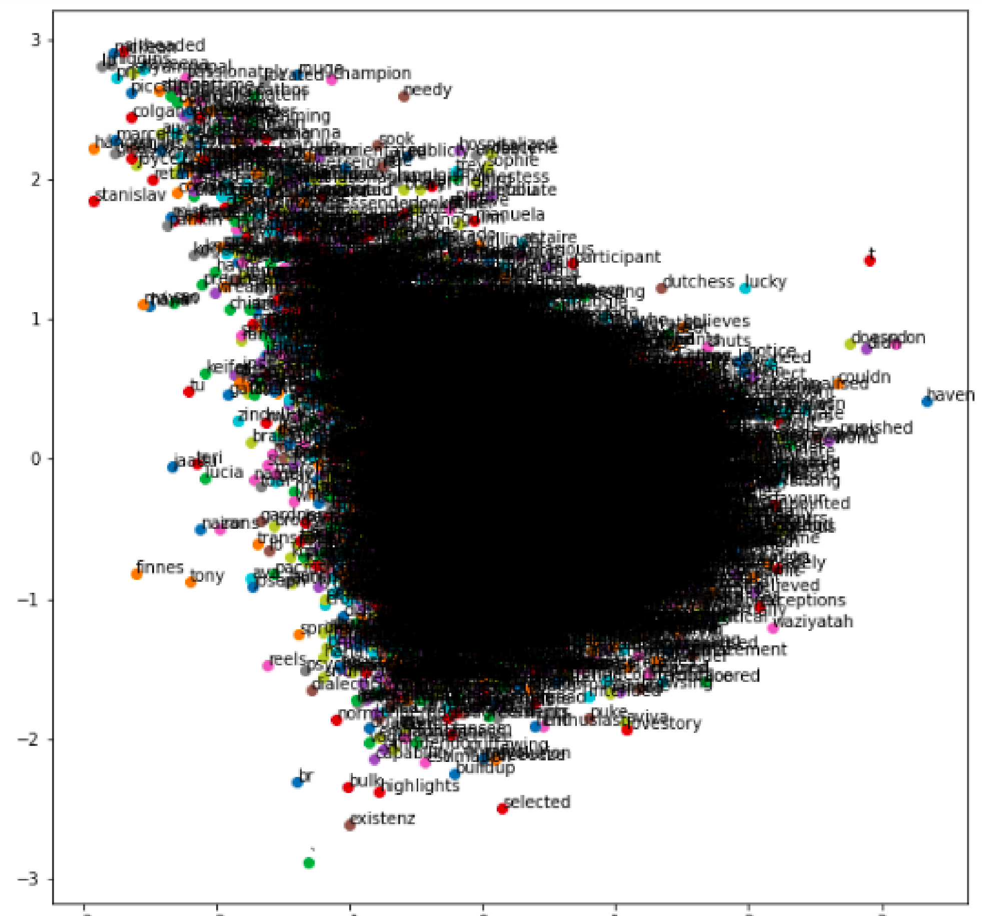

```
# Reading the text from the dataset
texts = pd.read_csv('data.txt', sep='\n')
texts = [x for x in texts['content'][:500]]
```

3. Since the dataset was very big , it was difficult to process it and implement and i was getting **IOPubbed data rate exceeded error** and my google colab crashed every time i tried to train my network.
   Solution : I used **keras data generator** to handle this big output by dividing the data into batches of size 400 which were called whenever my network trained itself.

```python
import numpy as np
import keras

class DataGenerator(keras.utils.Sequence):

    def __init__(self, word_list, unique_word_dict, batch_size=64,
                 n_classes=10, shuffle=True):

        self.word_list = word_list
        self.unique_word_dict = unique_word_dict
        self.n_words = len(unique_word_dict)
        self.batch_size = batch_size
        self.n_classes = n_classes
        self.shuffle = shuffle
        self.on_epoch_end()
```

```python
# Optimizing the network weights
model.fit_generator(
    generator=txtdata,
    epochs=50,
    use_multiprocessing=True,
    workers=10
    )
```

Result and Comparison :



Even after using a data generator to reduce the memory load, for full data, it projected 14 hr to complete one epoch and ultimately I had to size down my data to be able to compute and verify results with limited GPU and CPU power and time.

For the given result i used the following parameters :

Text Files of Dataset - 500

Epochs - 50

Batch size - 400

Time Taken - 2.5 Hours

Window Size - 2

Comparison and Insights :

Our current model involves one hot encoding vector which makes it **very expensive and memory consuming** and hence impractical for large dataset since for example for a text with **n words.** It creates vectors of **n^2** sizes which creates memory problems with increase of n. Sparsing those vectors in order to reduce their size makes the algorithm more stable for big dataset.

Conclusion :

Our current model involves one hot encoding vector which makes it very expensive and memory consuming.
The most commonly used models are **like GloVe** which are both unsupervised approaches based on the distributional hypothesis (words that occur in the same contexts tend to have similar meanings).While FastText is an improvement of Word2Vec uses the inclusion of **character n-grams,** which allows computing word representations for words that did not appear in the **training data ("out-of-vocabulary" words).** Unlike traditional word embeddings such as **word2vec and GLoVe, the ELMo** vector assigned to a token or a word depends on current context and is actually a function of the entire sentence containing that word. So the same word can have different word vectors under different contexts.

Future Works :

In a limited amount of time and computational resources, I was able to focus on a very narrow part of NLP. In the future I would like to explore better and state of the art word embeddings techniques like **GloVe,ELMo and FastText etc** . I need to learn approaches like **skip-gram, n-gram and working of ELMo** which allows for the same words to have different context which is more realistic. Some of the areas where I would like to work in NLP include - **Text Summarization, Sentiment Analysis and Neural Machine Translation etc.**