NATURAL LANGUAGE PROCESSING

TOPIC - PYTHON Project on CBOW

Name-Vibhuti Narayan Singh

Roll NO- BTECH/10501/21

BRANCH – CSE

SEC – B

SUBMITTED TO: INDRAJIT MUKHARJEE

INTRODUCTION

This is my Natural Language Processing project on the topic CBOW or Continious bag of words. Using a neural network with only a couple layers, word2vec tries to learn relationships between words and embeds them in a lower-dimensional vector space. To do this, word2vec trains words against other words that neighbor them in the input corpus, capturing some of the meaning in the sequence of words.A bag of words (BoW) is a representation of text that describes the occurrence of words within a text corpus, but doesn’t account for the sequence of the words. That means it treats all words independently from one another, hence the name bag of words.

BoW consists of a set of words (vocabulary) and a metric like frequency or term frequency-inverse document frequency (TF-IDF) to describe each word’s value in the corpus. That means BoW can result in sparse matrices and high dimensional vectors that consume a lot of computer resources if the vocabulary is very large.

To simplify the concept of BoW vectorization, imagine we have two sentences:

```
The dog is white
The cat is black
```

Converting the sentences to a vector space model would transform them in such a way that looks at the words in all sentences, and then represents the words in the sentence with a number. If the sentences were one-hot encoded:

```
The dog cat is white black
The dog is white = [1,1,0,1,1,0]
The cat is black = [1,0,1,1,0,1]
```

The BoW approach effectively transforms the text into a fixed-length vector to be used in machine learning.

Word2vec

Word2vec is a group of related models that are used to produce word embeddings. These models are shallow, two-layer neural networks that are trained to reconstruct linguistic contexts of words. Word2vec takes as its input a large corpus of text and produces a vector space, typically of several hundred dimensions, with each unique word in the corpus being assigned a corresponding vector in the space. Word vectors are positioned in the vector space such that words that share common contexts in the corpus are located close to one another in the space.

CBOW

CBOW or Continous bag of words is to use embeddings in order to train a neural network where the context is represented by multiple words for a given target words.

For example, we could use “cat” and “tree” as context words for “climbed” as the target word. This calls for a modification to the neural network architecture. The modification, shown below, consists of replicating the input to hidden layer connections C times, the number of context words, and adding a divide by C operation in the hidden layer neurons.

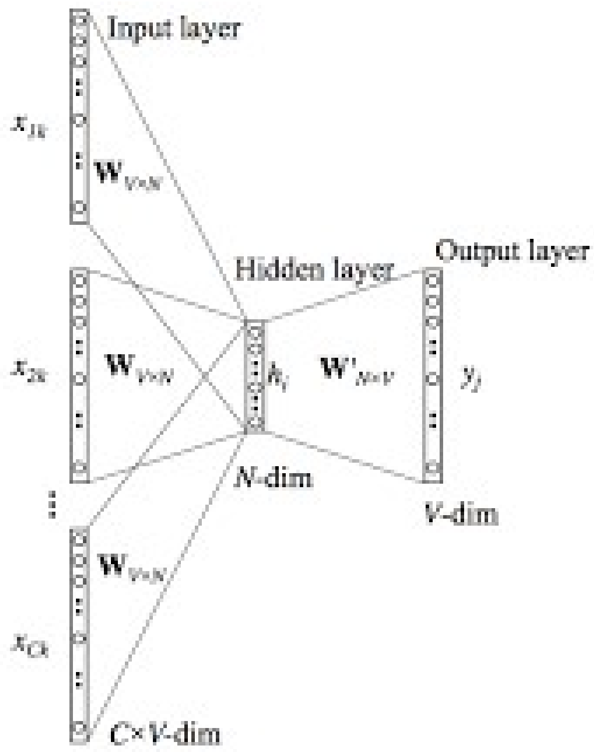
CBOW Architecture:

In [8]:

```
from IPython.display import Image
Image(filename="CBOW Architecture.png",width=400,height=400)

CBOW ARCHITECTURE:
```

Out[8]:



The CBOW architecture is pretty simple contains :

- the word embeddings as inputs (idx)
- the linear model as the hidden layer
- the log_softmax as the output

Applications of CBOW

Applications of CBOW in NLP:

```
Word Embeddings:
CBOW is used to generate word embeddings, which are dense vector repr
esentations of words. These embeddings capture semantic relationships
between words and are used in various NLP tasks such as text classifi
cation, sentiment analysis, and machine translation.

Semantic Similarity:
CBOW embeddings can be used to measure the semantic similarity betwee
n words. Words with similar meanings tend to have similar vector repr
```

Challenges of CBOW

Challenges of CBOW in NLP:

```
Loss of Word Order Information:
CBOW ignores the order of words in a sequence, treating them as an un
ordered set. This can be a limitation in tasks where word order is cr
ucial, such as in some language understanding or generation tasks.

Out-of-Vocabulary Words:
CBOW might struggle with out-of-vocabulary words, as it learns embedd
ings for the words in its training set. Words not present in the trai
ning set may not have meaningful representations, and handling rare w
ords can be a challenge.

Context Window Size:
The effectiveness of CBOW is sensitive to the choice of the context w
indow size. Too small a window may not capture enough context, while
too large a window may dilute the context. Finding the optimal window
size depends on the specific application.

Handling Polysemy:
Polysemy, where a word has multiple meanings, can be challenging for
CBOW. The model may struggle to capture the correct sense of a word i
n different contexts.

Computational Complexity:
Training CBOW models can be computationally intensive, especially whe
n dealing with large vocabularies and datasets. This complexity can b
e a bottleneck, particularly in resource-constrained environments.

Domain Specificity:
CBOW embeddings are trained on a specific corpus, and their performan
ce may suffer when applied to a domain different from the training da
ta. Fine-tuning or domain adaptation may be necessary for optimal per
formance in specialized domains.
```

Despite these challenges, CBOW remains a popular choice for certain NLP tasks, and researchers continue to explore variations and improvements to address its limitations.

DATASET

In [10]:
```
sentences = """We are about to study the idea of a computational process.
Computational processes are abstract beings that inhabit computers.
As they evolve, processes manipulate other abstract things called data.
The evolution of a process is directed by a pattern of rules
called a program. People create programs to direct processes. In effect,
we conjure the spirits of the computer with our spells."""
```

Clean Data

In [12]:
```
import re
# remove special characters
sentences = re.sub('[^A-Za-z0-9]+', ' ', sentences)

# remove 1 letter words
sentences = re.sub(r'(?:^| )\w(?:$| )', ' ', sentences).strip()

# lower all characters
sentences = sentences.lower()
```

Vocabulary

In [13]:
```
words = sentences.split()
vocab = set(words)
```

In [14]:
```
vocab_size = len(vocab)
embed_dim = 10
context_size = 2
```

Implementation

Dictionaries

In [15]:
```
word_to_ix = {word: i for i, word in enumerate(vocab)}
ix_to_word = {i: word for i, word in enumerate(vocab)}
```

Data Bags

In [16]:
```
# data - [(context), target]

data = []
for i in range(2, len(words) - 2):
    context = [words[i - 2], words[i - 1], words[i + 1], words[i + 2]]
    target = words[i]
    data.append((context, target))
print(data[:5])
```

```
[(['we', 'are', 'to', 'study'], 'about'), (['are', 'about', 'study', 'the'],
'to'), (['about', 'to', 'the', 'idea'], 'study'), (['to', 'study', 'idea', 'o
f'], 'the'), (['study', 'the', 'of', 'computational'], 'idea')]
```

Embeddings

In [19]:
```
import numpy as np
embeddings =  np.random.random_sample((vocab_size, embed_dim))
```

Linear Model

In [20]:
```
def linear(m, theta):
    w = theta
    return m.dot(w)
```

Log softmax + NLLloss = Cross Entropy

In [21]:
```
def log_softmax(x):
    e_x = np.exp(x - np.max(x))
    return np.log(e_x / e_x.sum())

def NLLLoss(logs, targets):
    out = logs[range(len(targets)), targets]
    return -out.sum()/len(out)

def log_softmax_crossentropy_with_logits(logits,target):

    out = np.zeros_like(logits)
    out[np.arange(len(logits)),target] = 1

    softmax = np.exp(logits) / np.exp(logits).sum(axis=-1,keepdims=True)

    return (- out + softmax) / logits.shape[0]


```

Forward Function

In [22]:
```
def forward(context_idxs, theta):
    m = embeddings[context_idxs].reshape(1, -1)
    n = linear(m, theta)
    o = log_softmax(n)

    return m, n, o
```

Backward Function

In [23]:
```
def backward(preds, theta, target_idxs):
    m, n, o = preds

    dlog = log_softmax_crossentropy_with_logits(n, target_idxs)
    dw = m.T.dot(dlog)

    return dw
```

Optimize Function

In [25]:
```
def optimize(theta, grad, lr=0.03):
    theta -= grad * lr
    return theta


```

Training

In [28]:
```
theta = np.random.uniform(-1, 1, (2 * context_size * embed_dim, vocab_size

epoch_losses = {}

for epoch in range(80):

    losses =  []

    for context, target in data:
        context_idxs = np.array([word_to_ix[w] for w in context])
        preds = forward(context_idxs, theta)

        target_idxs = np.array([word_to_ix[target]])
        loss = NLLLoss(preds[-1], target_idxs)

        losses.append(loss)

        grad = backward(preds, theta, target_idxs)
        theta = optimize(theta, grad, lr=0.03)


    epoch_losses[epoch] = losses


```

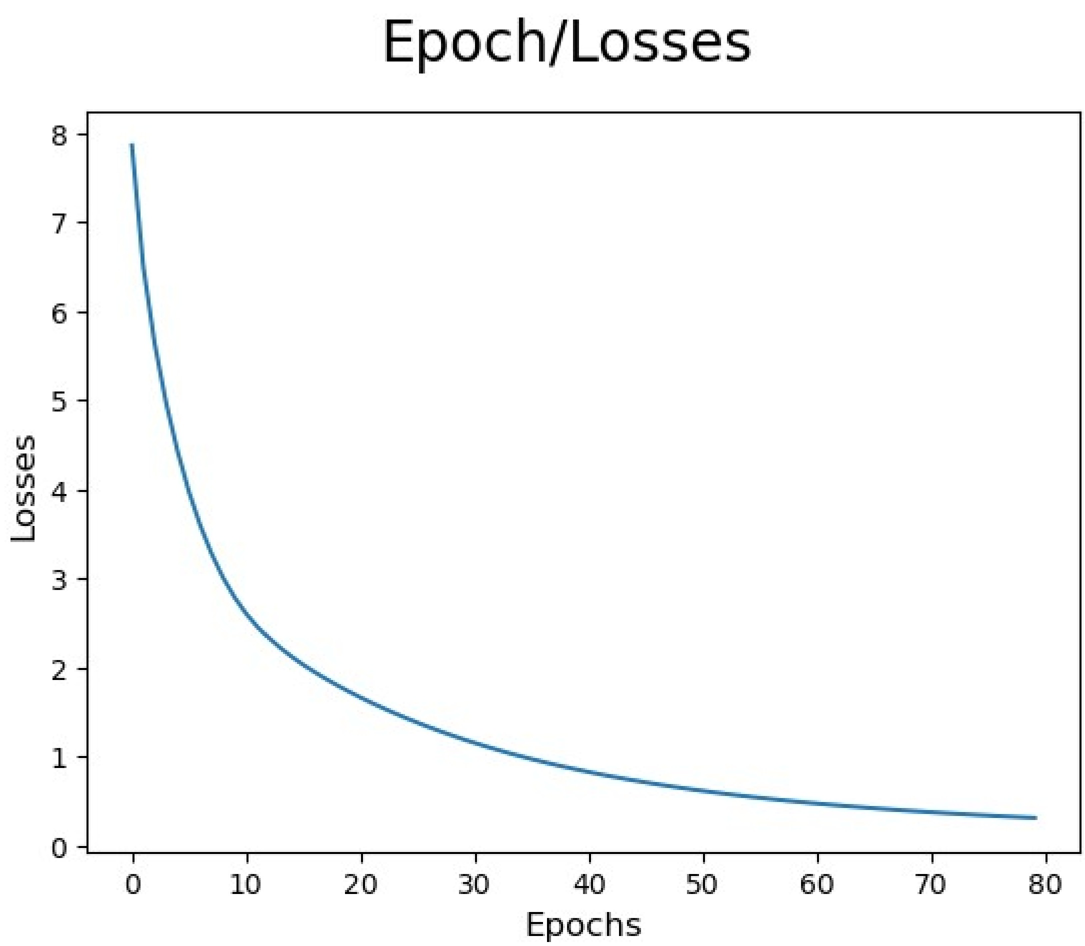
Analyze

Plot loss/epoch

In [29]:
```
import matplotlib.pyplot as plt
ix = np.arange(0,80)

fig = plt.figure()
fig.suptitle('Epoch/Losses', fontsize=20)
plt.plot(ix,[epoch_losses[i][0] for i in ix])
plt.xlabel('Epochs', fontsize=12)
plt.ylabel('Losses', fontsize=12)
```

Out[29]:
```
Text(0, 0.5, 'Losses')
```



Predict function

In [30]:
```python
def predict(words):
    context_idxs = np.array([word_to_ix[w] for w in words])
    preds = forward(context_idxs, theta)
    word = ix_to_word[np.argmax(preds[-1])]

    return word

# (['we', 'are', 'to', 'study'], 'about')
predict(['we', 'are', 'to', 'study'])


```

Out[30]: 'about'

Accuracy

In [31]:
```python
def accuracy():
    wrong = 0

    for context, target in data:
        if(predict(context) != target):
            wrong += 1

    return (1 - (wrong / len(data)))

accuracy()


```

Out[31]: 1.0