In-Class Assignment: Training Word Embeddings with CBOW

DATA 5420/6420

Name: Dallin Moore

In this notebook we will cover several models used for feature engineering word embeddings for text, so that we can better represent word meaning in our texts.

In this example we will use the *King James Bible* from the gutenberg corpus as a large sample that we can use to train a word-embedding model. Training word-embedding models is pretty computationally expesnive, and time-consuming. So, I'm going to show you how to access a GPU instance from google colab and you will run this notebook outside of class :)

While I want you to understand how word-embedding models can be trained, in the next template, we'll also play around with some pretrained models as well, which oftentimes can work quite well out of the box and can save time and effort. We'll apply those pretrained models to our toy corpus from P1 and then extend it to our document clustering.

1) Getting Set up

Let's get started with reading in our required libraries and packages:

```
import nltk
nltk.download('stopwords')
nltk.download('gutenberg')
nltk.download('punkt')
from nltk.corpus import gutenberg

from string import punctuation
from urllib import request
from bs4 import BeautifulSoup

import numpy as np
import pandas as pd
import re

import numpy as np

wpt = nltk.WordPunctTokenizer()                                          # assign wordpunctokenizer to wpt
stop_words = nltk.corpus.stopwords.words('english')                      # bring in stopwords from nltk
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package gutenberg to /root/nltk_data...
[nltk_data]   Package gutenberg is already up-to-date!
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

```
url = "https://gutenberg.org/files/10/10-h/10-h.htm"
response = request.urlopen(url)

raw = response.read().decode('utf-8-sig')
raw
print(raw.find("1:1"), ":", raw.rfind("END OF THE PROJECT GUTENBERG"))
```

```
9090 : 4707932
```

```
# raw = raw[10282:4634877]
raw = raw[9090:4707932]

raw[:400]
```

```
'1:1 In the beginning God created the heaven and the earth.\r\n</p>\r\n\r\n<p>\r\n1:2 A
nd the earth was without form, and void; and darkness was upon the face of\r\nthe deep.
And the Spirit of God moved upon the face of the waters.\r\n</p>\r\n\r\n<p>\r\n1:3 And
God said, Let there be light: and there was light.\r\n</p>\r\n\r\n<p>\r\n1:4 And God sa
```

A) Define Normalize Document Function

```
def normalize_document(doc):
    soup = BeautifulSoup(doc, 'html.parser')
    [s.extract() for s in soup(['iframe', 'script'])]
    doc = soup.get_text()
    doc = re.sub(r'[^a-zA-Z\.\s]', '', doc)
    doc = doc.lower()
    doc = doc.strip()
    tokens = wpt.tokenize(doc)
    filtered_tokens = [token for token in tokens if token not in stop_words]
    doc = ' '.join(filtered_tokens)
    return doc
```

B) Apply Normalization Function, tokenize the sentences, then remove punctation

```
norm_bible = normalize_document(raw)
norm_bible[:100]
```

```
'beginning god created heaven earth . earth without form void darkness upon face deep . spirit god mo'
```

```
punkt_st = nltk.tokenize.PunktSentenceTokenizer()
norm_sents = punkt_st.tokenize(norm_bible)
print(norm_sents)
def remove_punc(string):
    punc = '''!()-[]{};:'"\, <>./?@#$%^&*_~'''
    for ele in string:
        if ele in punc:
            string = string.replace(ele, " ")
    return string

norm_sents = [remove_punc(i) for i in norm_sents]
norm_sents[:5]
```

```
['beginning god created heaven earth .', 'earth without form void darkness upon face deep .', 'spirit god moved upon face waters .', 'go
['beginning god created heaven earth  ',
 'earth without form void darkness upon face deep  ',
 'spirit god moved upon face waters  ',
 'god said let light light  ',
 'god saw light good god divided light darkness  ']
```

2) Training a Word2Vec Model for Word Embeddings

Continuous Bag of Words (CBOW) Model

Let's first try implementing this method from scratch to understand how the model works; it involves five primary steps:

1. Building the corpus vocabulary
2. Building a CBOW (context, target) generator
3. Building the CBOW model archicture
4. Training the model
5. Getting the word embeddings!

A) Building corpus vocab

The first step in this process is separating out the text into the vocabulary, which consists of all the unique words in the text. We will need to bring in some new libraries that are often used in NLP and machine learning, including `tensorflow`.

```
!pip install np_utils
```

```
# from keras.preprocessing import text
# from keras.utils import np_utils
# from keras.preprocessing import sequence
# from tensorflow.keras.preprocessing.sequence import pad_sequences
```

```
from tensorflow.keras.preprocessing.text import Tokenizer, text_to_word_sequence
import np_utils
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.preprocessing.sequence import pad_sequences


tokenizer = Tokenizer()
tokenizer.fit_on_texts(norm_sents)                                                  # word tokenize norm_bible text
word2id = tokenizer.word_index                                                      # store word index numbers


word2id['PAD'] = 0
id2word = {v:k for k, v in word2id.items()}                                         # create dictionary of words with their index numb

wids = [[word2id[w] for w in text.text_to_word_sequence(doc)] for doc in norm_sents]      # create list of lists of word sequences for each
```

Let's take a step back and see what each of these variables contain...

```
print(norm_sents[0])
print(wids[0])
```

```
     beginning god created heaven earth
     [573, 6, 1189, 96, 46]
```

```
vocab_size = len(word2id)
embed_size = 100
window_size = 2
```

What does it mean to have a window size of 2?

It determines how far forwards and backwards the model looks to build the model.

> **B) Building CBOW (Context, Target) Generator** -- Add in comments to track what's going on here...

```
def generate_context_word_pairs(corpus, window_size, vocab_size):
    context_length = window_size*2 # context length of 4 words
    for words in corpus:
        sentence_length = len(words) # save the length of the sentence
        for index, word in enumerate(words):
            context_words = [] # store the context word
            label_word = [] # store the target word
            start = index - window_size # start at 2 words left of the target
            end = index + window_size + 1 # end 2 words to the right of the target

            context_words.append([words[i]
                                  for i in range(start, end)
                                  if 0 <= i < sentence_length
                                  and i != index]) # fill the context words for each word in the corpus from start to end
            label_word.append(word) # add the target word

            x = pad_sequences(context_words, maxlen = context_length) # save the context words
            y = np_utils.to_categorical(label_word, vocab_size) # save associated context word to y
            yield(x, y)
```

Now let's apply our context word pair generator to a sample of our first 10 words from our corpus to see a) what it does and b) that it worked.

```
from keras.utils import to_categorical
import np_utils

i = 0
for x, y in generate_context_word_pairs(corpus = wids, window_size=window_size, vocab_size=vocab_size):
    if 0 not in x[0]:
        print('Context (X):', [id2word[w] for w in x[0]], '-> Target (Y):', id2word[np.argwhere(y[0])[0][0]])
        if i == 9:
            break
        i += 1
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-76-af546a5d631c> in <cell line: 5>()
      3
      4 i = 0
----> 5 for x, y in generate_context_word_pairs(corpus = wids, window_size=window_size,
vocab_size=vocab_size):
      6     if 0 not in x[0]:
      7         print('Context (X):', [id2word[w] for w in x[0]], '-> Target (Y):',
id2word[np.argwhere(y[0])[0][0]])

<ipython-input-67-651aee90b1b4> in generate_context_word_pairs(corpus, window_size,
vocab_size)
     16
     17             x = pad_sequences(context_words, maxlen = context_length) # save
the context words
---> 18             y = np_utils.to_categorical(label_word, vocab_size) # save
associated context word to v
```

Let's check in a make sure we understand the components here: What is the window size and how do you know? What is the relationship between the context words (X) and the target word (Y)?

Because the window size was set to two, we see the two words before the target word, and the two words after the target word in the list of context words. For every value of y (a target word), there are a list of four values for x (context words).

C) Build the CBOW Architecture

This step is a brief venture into some deep learning to utilize Word2Vec to training word embeddings...The CBoW archictecture can be broken down into:

1. An embedding layer (where our context words are pass in as inputs) → word weights/embeddings are randomly initialized
2. A lambda layer → average the word embeddings
3. Dense softmax layer → predicts the target word

We match the predicted target against the actual target word and compute the error/loss using a metric called `categorical_crossentropy` then perform back propagation with each epoch (full run through) of our model...

We will then take the learned weights from this word prediction task and use them as our word embeddings!

```
import keras.backend as K
from keras.models import Sequential
from keras.layers import Dense, Embedding, Lambda


cbow = Sequential()
cbow.add(Embedding(input_dim = vocab_size, output_dim = embed_size, input_length = window_size*2))
cbow.add(Lambda(lambda x: K.mean(x, axis = 1), output_shape = (embed_size,)))
cbow.add(Dense(vocab_size, activation = 'softmax'))
cbow.compile(loss = 'categorical_crossentropy', optimizer = 'rmsprop')

print(cbow.summary())
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding (Embedding)       (None, 4, 100)            1256300

 lambda (Lambda)             (None, 100)               0

 dense (Dense)               (None, 12563)             1268863

=================================================================
Total params: 2525163 (9.63 MB)
Trainable params: 2525163 (9.63 MB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
None
```

```
print(vocab_size*100)
print(vocab_size*100+vocab_size)
```

```
1256300
1268863
```

Examining the output shape, do any of these values look familiar? Let's talk them through.

Embedding: There are 12563 words, in a matrix with 100 columns Lambda: this layer is an average of 4 words, so it is only 100 rows Dense: to make the prediction, an additional 12563 (the vocab size)

D) Train the model (for a few epochs)

So here's the step I'll have you guys do outside of class, because this is going to take some time to train. Click the *Runtime* menu above, and select the option to *Change runtime type* and select the *GPU* option. Once you do that, you'll need to rerun the above cells (it will start a new session) and then go ahead and kick off training. This might take like 2 hours or so...so go watch a movie or work on something else and leave this running. The joys of training complex models!

```
%%time

for epoch in range(1,4):
    loss = 0.
    i = 0
    for x, y in generate_context_word_pairs(corpus = wids, window_size=window_size, vocab_size=vocab_size):
        i +=1
        loss += cbow.train_on_batch(x, y)
        if i % 1000 == 0:
            print('Processed {} (context, word) pairs'.format(i))
    print('Epoch:', epoch, '\tloss:', loss)
    print()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<timed exec> in <module>

<ipython-input-67-651aee90b1b4> in generate_context_word_pairs(corpus, window_size,
vocab_size)
     16
     17             x = pad_sequences(context_words, maxlen = context_length) # save
the context words
---> 18             y = np_utils.to_categorical(label_word, vocab_size) # save
associated context word to y
     19             yield(x, y)
```

How many epochs did you try? How long did it take to train your word embeddings model (approximately)?

Double-click (or enter) to edit

D) Get the word embeddings!

First we need to grab the weights for each word, this is a stored parameter from `cbow`. We'll put these in a dataframe with the word labels.

```
# now follow this code

weights = cbow.get_weights()[0]                                          # assign the learned weights to weights
weights = weights[1:]                                                    # skip first token (start token)
print(weights.shape)

pd.DataFrame(weights, index=list(id2word.values())[1:]).head()           # show weights with an index of their associated word
```

Similar to how we examined the distance between document vectors in the previous assignment using cosine similarity to cluster similar documents, now we'll use another distance metric, euclidean distance, to find words with similar word vectors based on the weights above.

```
from sklearn.metrics.pairwise import euclidean_distances

distance_matrix = euclidean_distances(weights)                                  # create a distance matrix based on the distanc
```

```
print(distance_matrix.shape)
```

```
similar_words = {search_term: [id2word[idx]
                               for idx in distance_matrix[word2id[search_term]-1].
                               argsort()[1:6]+1]
                                  for search_term in ['god', 'jesus', 'noah',
                                                      'egypt', 'john', 'moses', 'famine']}    # find the top 5 words most similar to the follo
similar_words
```

Some of these make sense together, but many do not. What would be the pros and cons of training our word embeddings model for a greater number of epochs?

Double-click (or enter) to edit