Text factual query using Word2Vec with TensorFlow

Intro

We can take words in a language as independent set of characters and define their meaning individually. Even though that works for basic meaning, the relationship between each of the words are lost. So, we need to convert them into a space or representation that is easy to manipulate while inter-relational features are preserved. Vector representations are the most commonly used construct for this purpose in Mathematics. Thus, *word embedding* is a process of mapping of words (or phrases) from the vocabulary to vectors of real numbers. TensorFlow's Word2Vec (https://www.tensorflow.org/tutorials/word2vec) model is widely used for this purpose.

Suggested readings: ¶

- To understand better Skip-Gram Model the following tutorial is suggested
  - Word2Vec Tutorial - The Skip-Gram Model (http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/)
- To know about more complex and effective implementations of word2vec models see
  - Distributed Representations of Words and Phrases and their Compositionality (http://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf)

Now let's start by trying to convert three sentences to dense vectors. As explained by Word2Vec (https://www.tensorflow.org/tutorials/word2vec) that Distributional Hypothesis (https://en.wikipedia.org/wiki/Distributional_semantics#Distributional_Hypothesis) relies on the assumption that words appearing in the same context probably share the semantic meaning. Dense calculations compared to sparse calculations are more efficient so using vectors allow words of similar meaning to appear near each other. Go ahead and run the next cell to get started by importing the necessary libraries and setting the basic sentences for us to work with.

In [ ]:
```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
from sklearn.decomposition import PCA
from collections import defaultdict
%matplotlib inline


# Let's for example consider a simple way to map words from sentences into dense
# Let's make a table with words coocurrencies and then project vectors of all wor

s = ['Sky is blue', 'She is getting better', 'Everything is possible']
dic = defaultdict(dict)
for sent in s:
    words = sent.split()
    for w in words:
        for w2 in words:
            dic[w][w2]=1

df = pd.DataFrame(dic)
df.fillna(0, inplace=True)
df
```

You will see the vector space created by every word in each sentence. If they appear in the same sentence then the weight is 1. If they do not appear in the same sentence then it's at 0. The table basically gives relations between of the words given the 3 context sentences.

Now go ahead and run the next cell to collapse the words relationship into smaller dimensions so you'll see the clustering.

In [ ]:
```
res = PCA().fit_transform(df)

font = {'size'   : 15}
plt.rc('font', **font)
plt.figure(figsize=(7,7))
plt.scatter(res[:,0], res[:,1])
plt.axis('off')
for i, label in enumerate(df.columns):
    x, y = res[i,0], res[i,1]
    plt.scatter(x, y)
    annot = {'has': (1, 50), 'is': (1, 5)}
    plt.annotate(label, xy=(x, y),
                 xytext=annot.get(label,(1+i*2, 6*i)),
                 textcoords='offset points',
                   ha='right', va='bottom', )
```

Exercise 1

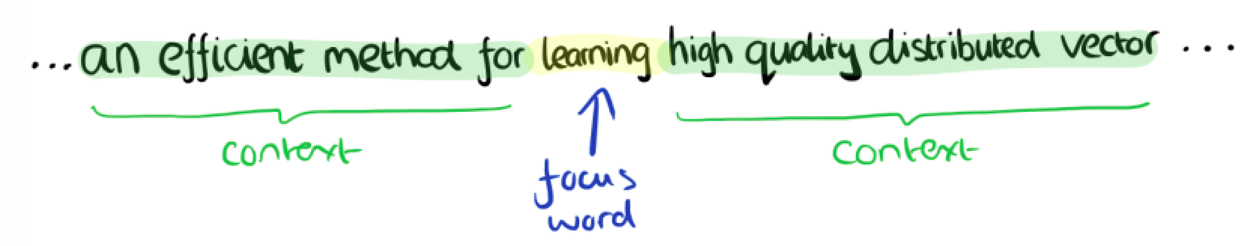
Q. Why is the word 'is' by itself?

```
A. Fill-in
```

word2vec: skip gram & cbow

Models **CBOW (Continuous Bag of Words)** and **Skip gram** were invented in the now distant 2013, *article*: [*Tomas Mikolov et al.* (https://arxiv.org/pdf/1301.3781v3.pdf)](https://arxiv.org/pdf/1301.3781v3.pdf)

- **CBOW** model predict missing word (focus word) using context (surrounding words).
- **skip gram** model is reverse to *CBOW*. It predicts context based on the word in focus.
- **Context** is a fixed number of words to the left and right of the word in focus (see picture below). The length of the context is defined by the "window" parameter.



Two models comparision

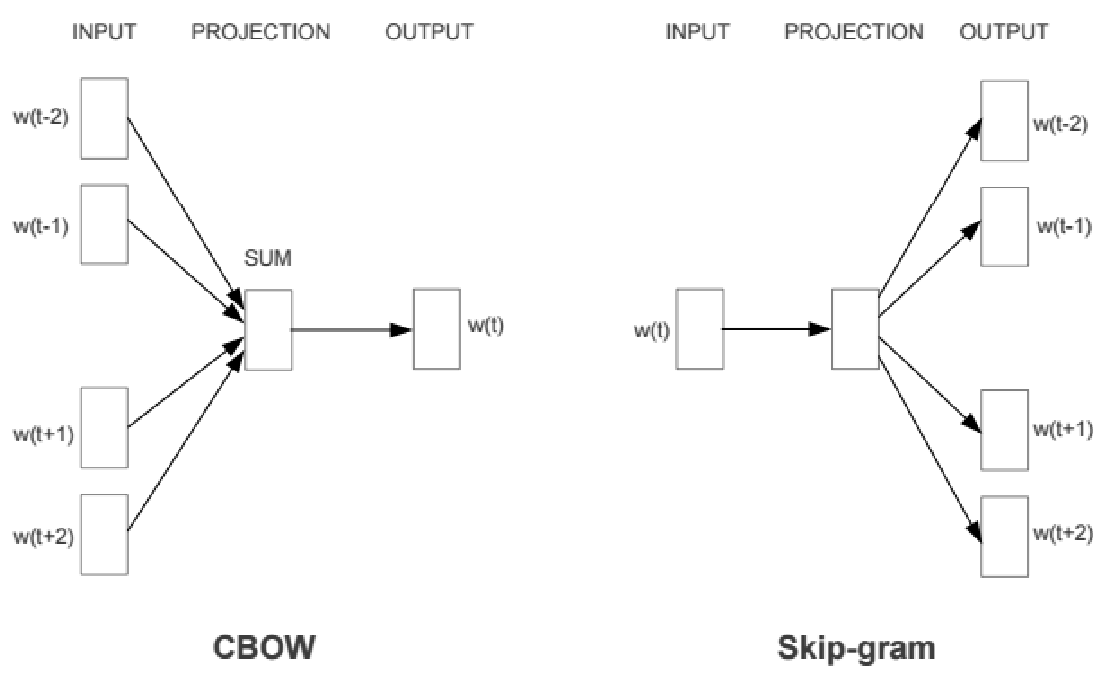


Figure 1: New model architectures. The CBOW architecture predicts the current word based on the context, and the Skip-gram predicts surrounding words given the current word.

There are a lot of implementations of word2vec e.g. [gensim (https://github.com/RaRe-Technologies/gensim/blob/develop/docs/notebooks/word2vec.ipynb)](https://github.com/RaRe-Technologies/gensim/blob/develop/docs/notebooks/word2vec.ipynb). And there are a lot of trained word-vectors which are already ready to use. But today we will learn how to create your own word embeddings.

Skip-gram

Consider a corpus with a sequence of words $w_1, w_2, .., w_T$.

Objective function (we would like to maximize it) for *skip gram* is defined as follow:

$$AverageLogProbability = \frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leqslant j\leqslant c, j\neq 0} log\ p(w_{t+j}|w_t)$$

- where c is a context length.
- w_t -- focus word

The basic formulation for probability $p(w_{t+j}|w_t)$ is calculated using **Softmax** -

$$p(w_o|w_i) = \frac{exp(s(v_i, v_o))}{\sum_{w=1}^{W} exp(s(v_w, v_i))}$$

where

- v_i and v_o input and output vector representations of w_i, w_o .
- $s(v_i, v_o) = v_o^T \cdot v_i$
- W is the number of words in vocabulary

CBOW

Predict word using context.

$$E = -log\ p(w_o\ |\ w_1,\ w_2,\ \dots,\ w_c)$$

The **probability** is the same as in the *skip gram* model, but now v_i is a sum of context-word vectors.

$$p(w_o\ |\ w_1,\ w_2,\ \dots,\ w_c) = \frac{exp(s(v_i, v_o))}{\sum_{w=1}^{W} exp(s(v_w, v_i))}$$

- $w_1,\ w_2,\ \dots,\ w_c$ -- input context words
- $v_i = \sum_{k=1}^{c} w_k$
- v_o = vector of output word
- $s(v_i, v_o) = v_o^T \cdot v_i$

Let's implement **`CBOW`** using tf framework.

And then implement **`skip gram`** using CBOW implementation as an example.

First import TensorFlow as we did with other libraries

In [ ]:
```
import tensorflow as tf
```

We will be using text8 dataset (http://mattmahoney.net/dc/textdata).

It's a 100 Mb dump of English Wikipedia site at the time of March 3, 2006. It gives us a rich dataset to work with while the size isn't too big yet.

Working with data

First we need to prepare the data so we can process it easily. One issue in NLP is the text corpus we have to deal with are usually long. Let's fetch the data but ensure that it's done only once.

In [ ]:
```
# WARNING! if this file "./data/text8.zip" doesn't exist
# it will be downloaded right now.

import os, urllib.request
def fetch_data(url):

    filename = url.split("/")[-1]
    datadir = os.path.join(os.getcwd(), "data")
    filepath = os.path.join(datadir, filename)

    if not os.path.exists(datadir):
        os.makedirs(datadir)
    if not os.path.exists(filepath):
        urllib.request.urlretrieve(url, filepath)

    return filepath

url = "http://mattmahoney.net/dc/text8.zip"
filepath = fetch_data(url)
print ("Data at {0}.".format(filepath))
```

Then unzip and read the data file.

In [ ]:
```
import os, zipfile

def read_data(filename):
    with zipfile.ZipFile(filename) as f:
        data = tf.compat.as_str(f.read(f.namelist()[0])).split()
    return data


words = read_data(filepath)
print("data_size = {0}".format(len(words)))
```

Only the first 50K more frequently used words are considered here N = 50000. The rest of the words are marked with unknow token "UNK".

In [ ]:
```
from collections import Counter

def build_dataset(words, vocabulary_size):
    count = [[ "UNK", -1 ]]
    count.extend(Counter(words).most_common(vocabulary_size-1))
    print("Least frequent word: ", count[-1])
    word_to_index = { word: i for i, (word, _) in enumerate(count) }
    data = [word_to_index.get(word, 0) for word in words] # map unknown words to
    unk_count = data.count(0) # Number of unknown words
    count[0][1] = unk_count
    index_to_word= dict(zip(word_to_index.values(), word_to_index.keys()))

    return data, count, word_to_index, index_to_word

vocabulary_size = 50000
data, count, word_to_index, index_to_word = build_dataset(words, vocabulary_size)

# Everything you need to know about the dataset

print("data: {0}".format(data[:5]))
print("count: {0}".format(count[:5]))
print("word_to_index: {0}".format(list(word_to_index.items())[:5]))
print("index_to_word: {0}".format(list(index_to_word.items())[:5]))
```

Exercise 2

Let's look at characteristics of the wiki dataset.

Q. How many 'unknown' occurrences are there?

```
A. Fill-in
```

Q. Which word has the highest occurrences?

```
A. Fill-in
```

Q. What is the index for that word?

```
A. Fill-in
```

Now that we know the characteristics of the dataset, let's batch up the data and start processing.

In [ ]:
```
import numpy as np
from collections import deque

def generate_batch(data_index, data_size, batch_size, bag_window):
    span = 2 * bag_window + 1 # [ bag_window, target, bag_window ]
    batch = np.ndarray(shape = (batch_size, span - 1), dtype = np.int32)
    labels = np.ndarray(shape = (batch_size, 1), dtype = np.int32)

    data_buffer = deque(maxlen = span)

    for _ in range(span):
        data_buffer.append(data[data_index])
        data_index = (data_index + 1) % data_size

    for i in range(batch_size):
        data_list = list(data_buffer)
        labels[i, 0] = data_list.pop(bag_window)
        batch[i] = data_list

        data_buffer.append(data[data_index])
        data_index = (data_index + 1) % data_size
    return data_index, batch, labels


print("data = {0}".format([index_to_word[each] for each in data[:16]]))
data_index, data_size, batch_size = 0, len(data), 4
for bag_window in [1, 2]:
    _, batch, labels = generate_batch(data_index, data_size, batch_size, bag_winc
    print("bag_window = {0}".format(bag_window))
    print("batch = {0}".format([[index_to_word[index] for index in each] for each
    print("labels = {0}\n".format([index_to_word[each] for each in labels.reshape
```

And now just take a close look at the output.

- We just want to implement *CBOW*, and therefore missed words are considered as the `labels`.
- Remember about the **window** parameter discussed above, here it is **`bag_window`**.
- Each sample in the batch has a number of words equal to **`bag_window * 2`**

CBOW architecture

Let's start implementing CBOW architecture. Pay close attention to each of the steps as you'll be having to replicate the process for skip-gram model. The two processes should be very similar to each other.

In [ ]:
```
import math

# define constants
batch_size = 256
embedding_size = 128

# How many words to consider from each side
bag_window = 2

tf.reset_default_graph()
graph = tf.Graph()
with graph.as_default():
    #
    # Take the vectors for the context words, which are all bag_window * 2
    train_data = tf.placeholder(tf.int32, [batch_size, bag_window * 2])
    # Label -- is a word in focus
    train_labels = tf.placeholder(tf.int32, [batch_size])

    #
    # Create an embedding matrix
    # and initialize it by sampling from the uniform distribution [-1, 1]

    embeddings = tf.Variable(tf.random_uniform([vocabulary_size, embedding_size],

    #
    # Get vectors corresponding to the indices of context words
    # embed is a matrix with shape [batch_size, bag_window * 2, embedding_size]
    embed = tf.nn.embedding_lookup(embeddings, train_data)

    # Sum up all the context vectors to the one vector with the same dimension
    # Here we got a matrix of such vectors with the shape [batch_size, embedding_
    context_sum = tf.reduce_sum(embed, 1)

    scores = tf.matmul(context_sum, embeddings, transpose_b=True)

    one_hot_labels = tf.one_hot(train_labels, vocabulary_size)
    loss_tensor = tf.losses.softmax_cross_entropy(onehot_labels=one_hot_labels, l
    loss = tf.reduce_mean(loss_tensor)


    optimizer = tf.train.AdamOptimizer(0.01).minimize(loss)

    # We need to normalize word embeddings for dot product to be a cosine distanc
    norm = tf.sqrt(tf.reduce_sum(tf.square(embeddings), 1, keep_dims = True))
    normalized_embeddings = embeddings / norm
```

Training CBOW model

For 8k steps on 2 physical CPU cores it takes about 18 minutes.

Unfortunately, it is not enough to train good embeddings.

Luckily, we have GPU available in this lab, so you could try to achieve reasonable quality much faster. Good representations should take over 50k iterations. Now go ahead and run it to train the model.

In [ ]:
```
%%time
#num_steps = 50001
num_steps = 8001

with tf.Session(graph=graph) as sess:
    try:
        tf.global_variables_initializer().run()
        print('Initialized')
        average_loss = 0

        for step in range(num_steps):
            data_index, batch, labels = generate_batch(data_index, data_size, bat
            feed_dict = { train_data: batch, train_labels: labels.reshape(-1) }
            _, current_loss = sess.run([optimizer, loss], feed_dict = feed_dict)
            average_loss += current_loss
            if step % 10 == 0:
                if step > 0:
                    average_loss = average_loss / 10
                    print ("step = {0}, average_loss = {1}".format(step, average_
                    average_loss = 0
    except KeyboardInterrupt:
        final_embeddings = normalized_embeddings.eval()
    final_embeddings = normalized_embeddings.eval()
```

Visualization

We can use projector (http://projector.tensorflow.org/) to visualize the results in a word cloud. Go ahead and click on projector (http://projector.tensorflow.org/) to launch the visualization.

Click on Projector (http://projector.tensorflow.org/)

In [ ]:
```
with open('embeddings.txt', 'w') as f:
    for n in range(vocabulary_size):
        s = '\t'.join([index_to_word[n]] + [str(num) for num in final_embeddings[
        f.write(s + '\n')
with open('mentadata.txt', 'w') as f:
    for n in range(vocabulary_size):
        f.write(index_to_word[n] + '\n')
```

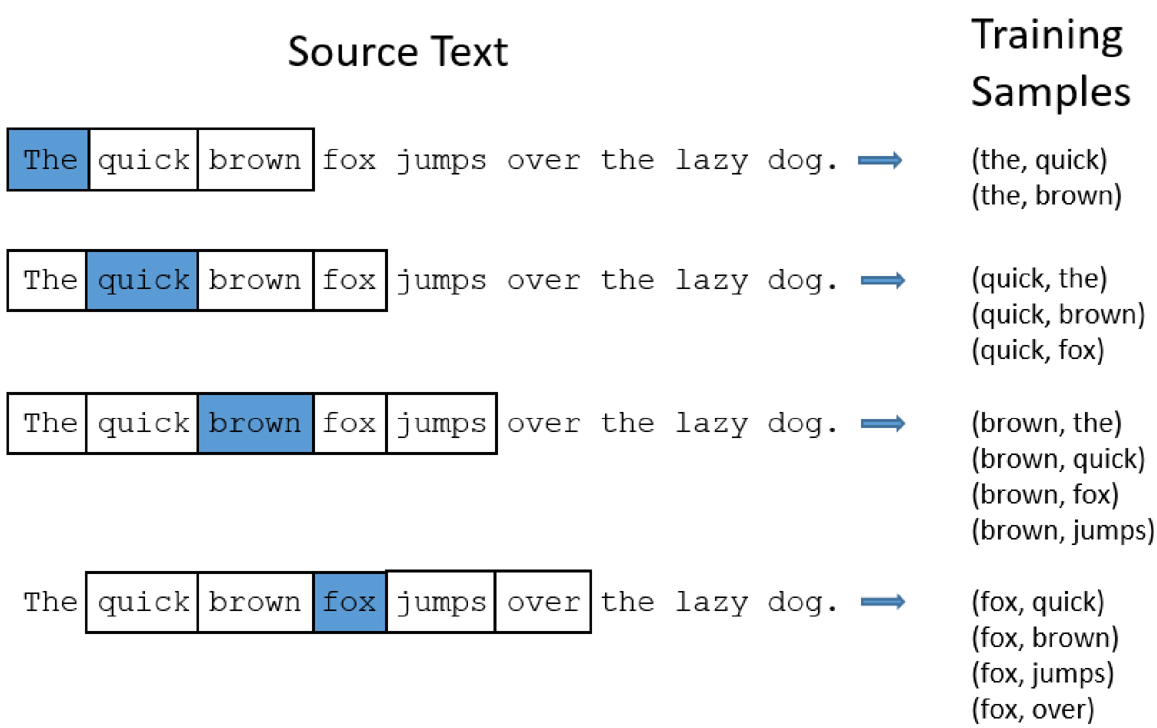
Or see TSNE here:

In [ ]:
```
from sklearn.manifold import TSNE

num_points = 350

tsne = TSNE(perplexity=10, n_components=2, init="pca", n_iter=5000)
two_d_embeddings = tsne.fit_transform(final_embeddings[1:num_points+1, :])

plt.figure(figsize=(15,15))
words = [index_to_word[i] for i in range(1, num_points+1)]

for i, label in enumerate(words):
    x, y = two_d_embeddings[i, :]
    plt.scatter(x, y)
    plt.annotate(label, xy=(x, y), xytext=(5, 2), textcoords="offset points",
                 ha="right", va="bottom")
plt.show()
```

Task

Your task is to implement `skip-gram` model, using code above.

This approach is nicely illustrated with this figure:



As you can see on the picture, the training set consists of pairs (`central word`, `context word`).

I.e. our model takes `central word` and should produce class in softmax, which corresponds to `context word`.

The difference between two models is not that big after all, so good luck with coding!

In [ ]:

```
# We have implemented batch generator for you
from collections import deque
import numpy as np
import random

def generate_batch_2(data_index, data_size, batch_size, num_skips, skip_window):
    assert batch_size % num_skips == 0
    assert num_skips <= 2 * skip_window

    batch = np.ndarray(shape = batch_size, dtype = np.int32)
    labels = np.ndarray(shape = (batch_size, 1), dtype = np.int32)
    span = 2 * skip_window + 1
    data_buffer = deque(maxlen = span)
    for _ in range(span):
        data_buffer.append(data[data_index])
        data_index = (data_index + 1) % data_size

    for i in range(batch_size // num_skips):
        target, targets_to_avoid = skip_window, [skip_window]
        for j in range(num_skips):
            while target in targets_to_avoid:
                target = random.randint(0, span - 1)
            targets_to_avoid.append(target)
            batch[i * num_skips + j] = data_buffer[skip_window]
            labels[i * num_skips + j, 0] = data_buffer[target]
        data_buffer.append(data[data_index])
        data_index = (data_index + 1) % data_size
    return data_index, batch, labels


print ("data = {0}\n".format([index_to_word[each] for each in data[:32]]))
data_index, data_size = 0, len(data)
for num_skips, skip_window in [(2, 1), (4, 2)]:
    data_index = 0
    data_index, batch, labels = generate_batch_2(data_index=data_index,
                                                 data_size=data_size,
                                                 batch_size=16,
                                                 num_skips=num_skips,
                                                 skip_window=skip_window)
    print ("data_index = {0}, num_skips = {1}, skip_window = {2}".format( data_ir
    print ("batch = {0}".format([index_to_word[each] for each in batch]))
    print ("labels = {0}\n".format([index_to_word[each] for each in labels.reshap
```

Now that we have shown how to implement CBOW model, you have all the tools to create skip-gram model yourself. In the following cell, create skip-gram model by following the steps from CBOW implementation.

Exercise 3

Fill-in the following cell with TensorFlow code for skip-gram model.

In [ ]:

Exercise 4

Once the model is created, next step is train it. Fill-in Tensorflow code for training of skip-gram model.

In [ ]:

```

```

Exercise 5

Finally, after training we want to visualize the results. Implement tSNE visualization of embeddings for skip-gram model.

In [ ]:

```

```

Real example

Let's test our embeddings on some real case: we create simple Wikipedia search engine.

To do that first of all we need to download Wikipedia sample:

In [ ]:

```
! wget -c "https://s3.amazonaws.com/fair-data/starspace/wikipedia_devtst.tgz"
```

In [ ]:

```
! tar -xzvf wikipedia_devtst.tgz
```

In [ ]:

```
! head wikipedia_test_basedocs.txt -1
```

Now we will vectorize all the articles:

In [ ]:

```
def vectorize(text):
    tokens = text.lower().split()
    num_words = 0
    doc_vector = np.zeros_like(final_embeddings[0])
    for token in tokens:
        if token in word_to_index:
            num_words += 1
            doc_vector += final_embeddings[word_to_index[token]]
    doc_vector /= num_words
    return doc_vector
```

In [ ]:
```
vectorized_docs = []
docs = []
with open("wikipedia_test_basedocs.txt",encoding='utf8') as f:
    for doc in f:
        doc_vector = vectorize(doc)
        vectorized_docs.append(doc_vector)
        docs.append(doc)
```

Our toy search engine:

In [ ]:
```
def search(query):
    query_vector = vectorize(query)
    ranking = []
    for i in range(len(vectorized_docs)):
        score = np.dot(query_vector, vectorized_docs[i])
        ranking.append((score, i))
    ranking.sort(key=lambda x: -x[0]) # to have descending sorting
    return docs[ranking[0][1]]
```

And at last we could test it on some query:

In [ ]:
```
query = "japanese strong gull"
```

In [ ]:
```
search(query)
```

Exercise 6

Create your own query and determine the quality of the results.

You could spot that results are not the best, so you could improve quality of the embeddings, by increasing windows size in training, increasing batch size and train longer.

In [ ]:
```

```