Word Embeddings

This article gives an introduction to word embeddings, it's advantages over other regular representations. We have discussed building word embeddings from scratch on imdb data and visualizing these vectors on a Tensorflow Embedding projector . So, let's get started.

Machine learning models in any field computer vision, text processing, audio analysis takes vectors of arrays of numbers as input. So, first preprocessing of data needs conversion of strings of words to numbers. Let's look at some basic strategies and their disadvantages before learning word embeddings..

One-hot Encodings

The vocabulary of our corpus is a set of unique words in the corpus. To represent each word create a zero vector of length equal to vocabulary and change the index corresponding to the word to one. This process is shown in below table

Corpus = ["Padhai is the best"]				
Padhai	1	0	0	0
is	0	1	0	0
best	0	0	0	1

Then we concatenate all one-hot encoded words of a sentence. This inefficient representation has several disadvantages. First, the vectors are very sparse, over 99.9% of elements remain zero for a large vocabulary. Second, it doesn't capture any relationship between similar or dissimilar words. For example words 'disappointing' and 'exceptional' are encoded the same way.

Word Tokenizer

This encodes each word with a unique number. FIrst it finds the number of unique words and assigns a number to each. For example the sentence "PadAI is the

best" can be encoded as a dense vector like [4,1,3,2]. This solves the problem of having sparse representations full of zeros. The downsides however this do not capture the relationship between words as the integer encoding is arbitrary. Now we'll see how to use the word tokenizer of keras for encoding to unique integers.

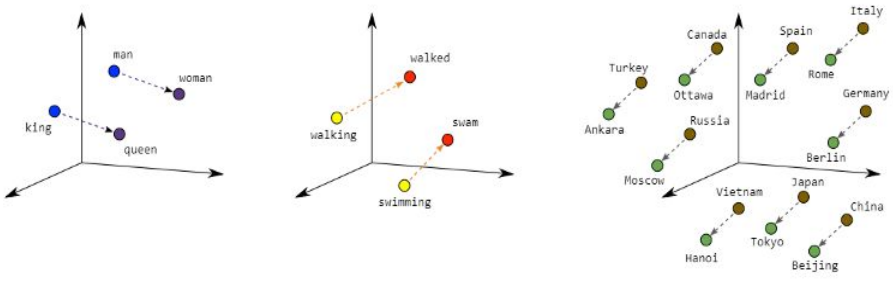
```
from tensorflow.keras.preprocessing.text import Tokenizer

corpus = ["ML is awesome!","Deep Learning is just updating random stuff fast enough"]
tokenizer = Tokenizer(num_words=15) #Considers only top common 15 words in the corpus
tokenizer.fit_on_texts(corpus)
word_index = tokenizer.word_index
print(word_index)
```

Output:

```
{'is': 1, 'ml': 2, 'awesome': 3, 'deep': 4, 'learning': 5, 'just': 6, 'updating': 7, 'random': 8, 'stuff': 9, 'fast': 10, 'enough': 11}
```

Using Word Embeddings

Word Embeddings translate large one-hot sparse vectors into a lower-dimensional space preserving semantic relationships. It maps large sparse/integer encodings to small dense vectors. The dimension of the embedding layer is a hyper parameter that needs to be optimized just like the number of neurons in a Dense layer. The figure below explains how in the vector space semantically similar words are fitted close to each other.



Male-Female Verb Tense Country-Capital

source: [Embeddings: Translating to a Lower-Dimensional Space]

When the embedding layer is created, the weights are randomly initialized and adjusted via back propagation during training. After training, similar words will have embedding encoded vectors closer to each other. Let's now see a simple embedding transformation of a vector in keras.

```
import tensorflow as tf

e = tf.keras.layers.Embedding(1000,7) #(Input_corpus_size,Embedding_output_dim)
embed_vec = e(tf.constant([1,2,3]))
print(embed_vec)
```

Output:

```
tf.Tensor(
[[-0.03179647 -0.02784987  0.03708848  0.00253868  0.03263071 -0.0065117
   0.01713542]
 [-0.01115904  0.00929461 -0.04185855  0.0455729  -0.03373976  0.00153339
   0.00735169]
 [ 0.00957887 -0.03844402 -0.01524512 -0.01141509  0.00257739 -0.02377379
  -0.0065446 ]], shape=(3, 7), dtype=float32)
```

There are two ways to get word embeddings, we will discuss first approach next

- Learn word embeddings for a problem specific data, task(eg: sentiment analysis). It assigns random numbers and later learns via back propagation during training.
- Load pre trained word embeddings(eg: word2vec, GloVe) that were precomputed for a different machine learning task into your model. This property of transfer learning makes word embeddings more powerful.

Sentiment analysis on IMDB data using Word Embeddings

Let's test the above learned concepts by performing a sentiment analysis on imdb reviews by using word embedding layer. Finally we'll load the learned embedding vectors into a tensorflow projector (Embedding projector - visualization of high-dimensional data) and check if semantic similar words are closer to each other or not.

First, download and load the data from tensorflow-datasets. Divide the corpus into train and test sections.

```
import tensorflow as tf
import tensorflow_datasets as tfds
import numpy as np

#Loading Data
imdb, info = tfds.load("imdb_reviews", with_info=True, as_supervised=True)
train_data, test_data = imdb['train'], imdb['test']

training_sentences = []
training_labels = []
testing_sentences = []
testing_labels = []

#Extracting text, labels from train_data, test_data
for s,l in train_data:
  training_sentences.append(s.numpy().decode('utf8'))
  training_labels.append(l.numpy())

for s,l in test_data:
  testing_sentences.append(s.numpy().decode('utf8'))
  testing_labels.append(l.numpy())

#Converting to numpy arrays
training_sentences = np.array(training_sentences)
training_labels = np.array(training_labels)

testing_sentences = np.array(testing_sentences)
testing_labels = np.array(testing_labels)

print(training_sentences.shape,testing_sentences.shape)
print(training_sentences[0])
print(training_labels[0])
```

```
(25000,) (25000,)
This was an absolutely terrible movie. Do not be lured in by Christopher Walken or Michael Ironside. Even their great acting could not redeem this movie ridiculous storyline. This movie is an early nineties US propaganda piece. The most pathetic scenes were those when the Columbian rebels were making their cases for revolutions. I am disappointed that there are movies like this, ruining actor like Christopher Walken. I could barely sit through it.
```

```
0
```

Let's tokenize the entire corpus into arbitrary integers and to make sure all the training data of same length we use padding on smaller sentences. Here we are mapping 10000 dim to just 16(embedding layer) dim .

```
vocab_size = 10000
embedding_dim = 16
max_length = 120 #Just considering first 120 words for every review
trunc_type = 'post'
oov_tok = "<OOV>" #Any new words not in vocab_size will be denoted as <OOV>

from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences

tokenizer = Tokenizer(num_words=vocab_size, oov_token=oov_tok)
tokenizer.fit_on_texts(training_sentences)

#Stores words, index in a dictionary
word_index = tokenizer.word_index

#Integer encoding and padding
sequences = tokenizer.texts_to_sequences(training_sentences)
padded = pad_sequences(sequences,maxlen=max_length,truncating=trunc_type)

testing_sequences = tokenizer.texts_to_sequences(testing_sentences)
testing_padded = pad_sequences(testing_sequences,maxlen=max_length)
```

It will be helpful to store the index of word and word itself in a dictionary. This is done using reverse_word_index.

```
reverse_word_index = dict([(value,key) for (key,value) in word_index.items()])
```

Now, create a simple model with an embedding layer, hidden dense layer and an output layer. Compile and run it for 10 epochs, now extract the weights of trained embedding layers for every word in word_index and store it for loading into a tensorflow projector. So, our embedding layer has 10000 x 16 weights to learn during training.

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim,
input_length=max_length),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(6, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
model.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])

num_epochs = 10
model.fit(padded, training_labels, epochs=num_epochs,
validation_data=(testing_padded, testing_labels))
```

```
#Extracting weights of embedding layer
e = model.layers[0]
weights = e.get_weights()[0]
print(weights.shape)
```

```
Output: (10000, 16)
```

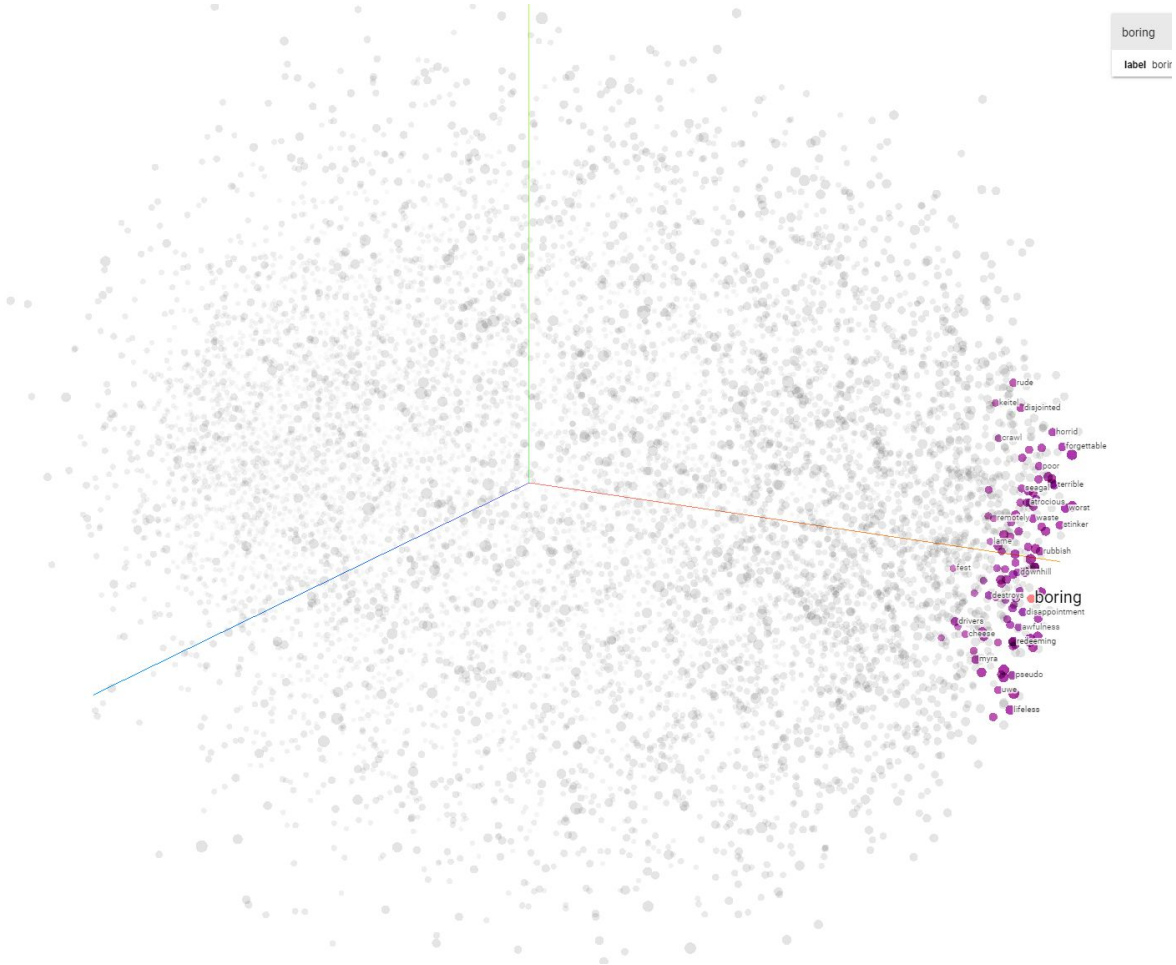
Writing these corresponding words, weights into meta.tsv, vecs.tsv files. The projector accepts files in a specific format so make sure you check that out on the website.

```
import io
from google.colab import files

out_v = io.open('vecs.tsv','w',encoding='utf-8')
out_m = io.open('meta.tsv','w',encoding='utf-8')

for word_num in range(1,vocab_size):
  word = reverse_word_index[word_num] #Extracting word for given index
  embeddings = weights[word_num] #Getting weights of 16 dim
  out_m.write(word+"\n")
  out_v.write('\t'.join([str(x) for x in embeddings]) + "\n") #Writing weights
out_v.close()
out_m.close()

files.download('vecs.tsv')
```

```
files.download('meta.tsv')
```

After uploading the two downloaded files, we can check the embedded vectors for all 10,000 words. Notice that semantically opposite words have vectors far as possible from each other and similar words are closer to each other (awesome, masterpiece, delight, underrated, great, wonderful etc).

On the opposite side words such as boring, rubbish, worst, rude, horrid, poor are closer to each other.



Conclusion

Word embeddings are considered as one of the successful applications of unsupervised learning at present. They use a very-low dimensional space while preserving semantic relationships. In this we have only looked at the first way of using word embeddings that is learning from training data. Next we shall talk about word2vec, GloVe representations and how to load and use them in our model until then enjoy machine learning !!