M   🔍   🔔   [Upgrade]   👤

Towards Data Science

Sharing concepts, ideas, and codes.

[Follow]

👏 3.2K

🔖

Learn Word2Vec by implementing it in tensorflow

👤 aneesh joshi [Follow]
Jul 9, 2017 · 10 min read

Hi!

I feel that the best way to understand an algorithm is to implement it. So, in this article I will be teaching you Word Embeddings by implementing it in Tensor Flow.

The idea behind this article is to avoid all the introductions and the usual chatter associated with word embeddings/word2vec and jump straight into the meat of things. So, much of the king-man-woman-queen examples will be skipped.

How do we make these embeddings?

There are **many** techniques to get word embeddings, we will discuss one technique which has gained a lot of fame, the one and only, word2vec. Contrary to popular belief, word2vec is not a *deep* network, it only has 3 layers!

Note : word2vec has a lot of technical details which I will skip over to make the understanding a lot easier.

How word2vec works:

The idea behind word2vec is that:

1. Take a 3 layer neural network. (1 input layer + 1 hidden layer + 1 output layer)
2. Feed it a word and train it to predict its neighbouring word.
3. Remove the last (output layer) and keep the input and hidden layer.
4. Now, input a word from within the vocabulary. The output given at the hidden layer is the '*word embedding*' of the input word.

That's it! Just doing this simple task enables our network to learn interesting representations of words.

Let's start implementing this to flesh out this understanding.

(the full code is available here . I suggest you understand what's going on from this article and then use the code from there.)

This is the raw text on which we will be working on:

(I have intentionally spaced out the periods and avoided them in the end to simplify. Once you understand, feel free to use a tokenizer)

```
import numpy as np
import tensorflow as tf
corpus_raw = 'He is the king . The king is royal . She is the royal
queen '
# convert to lower case
corpus_raw = corpus_raw.lower()
```

Towards Data Science

Sharing concepts, ideas, and codes.

Follow

👏 3.2K

🔖

Samples

	Samples
The **quick** brown fox jumps over the lazy dog. ➡	(the, quick) (the, brown)
The **quick** brown fox jumps over the lazy dog. ➡	(quick, the) (quick, brown) (quick, fox)
The quick **brown** fox jumps over the lazy dog. ➡	(brown, the) (brown, quick) (brown, fox) (brown, jumps)
The quick brown **fox** jumps over the lazy dog. ➡	(fox, quick) (fox, brown) (fox, jumps) (fox, over)

A training sample generation with a window size of 2.

Note: If the word is at the beginning or ending of sentence, the window ignores the outer words.
Before doing this, we will create a dictionary which translates words to integers and integers to words. This will come in handy later.

```
words = []
for word in corpus_raw.split():
    if word != '.': # because we don't want to treat . as a word
        words.append(word)
words = set(words) # so that all duplicate words are removed
word2int = {}
int2word = {}
vocab_size = len(words) # gives the total number of unique words
for i,word in enumerate(words):
    word2int[word] = i
    int2word[i] = word
```

What these dictionaries allow us to do is :

```
print(word2int['queen'])
-> 42 (say)
print(int2word[42])
-> 'queen'
```

Now, we want a list of our sentences as a list of words:

```
# raw sentences is a list of sentences.
raw_sentences = corpus_raw.split('.')
sentences = []
for sentence in raw_sentences:
    sentences.append(sentence.split())
```

This will give us a list of sentences where each sentence is a list of words.

```
print(sentences)
-> [['he', 'is', 'the', 'king'], ['the', 'king', 'is', 'royal'],
['she', 'is', 'the', 'royal', 'queen']]
```

Now, we will generate our training data:
(This might get difficult to read within medium. refer to the code link)

```
data = []
WINDOW_SIZE = 2
for sentence in sentences:
    for word_index, word in enumerate(sentence):
        for nb_word in sentence[max(word_index - WINDOW_SIZE, 0) :
min(word_index + WINDOW_SIZE, len(sentence)) + 1] :
            if nb_word != word:
                data.append([word, nb_word])
```

This basically gives a list of word, word pairs. *(we are considering a window size of 2)*

```
print(data)
[['he', 'is'],
 ['he', 'the'],
 ['is', 'he'],
 ['is', 'the'],
 ['is', 'king'],
 ['the', 'he'],
 ['the', 'is'],
```

DATA SCIENCE   MACHINE LEARNING   PROGRAMMING   VISUALIZATION   AI   JOURNALISM   EVENTS   SUBMIT

Towards Data Science

Sharing concepts, ideas, and codes.

[Follow]

👏 3.2K

🔖

We have our training data. But it needs to be represented in a way a computer can understand i.e., with numbers. That's where our `word2int` dict comes handy.

Let's go one step further and convert these numbers into one hot vectors.

```
i.e.,
say we have a vocabulary of 3 words : pen, pineapple, apple
where
word2int['pen'] -> 0 -> [1 0 0]
word2int['pineapple'] -> 1 -> [0 1 0]
word2int['apple'] -> 2 -> [0 0 1]
```

Why one hot vectors? : told later

```python
# function to convert numbers to one hot vectors
def to_one_hot(data_point_index, vocab_size):
    temp = np.zeros(vocab_size)
    temp[data_point_index] = 1
    return temp
x_train = [] # input word
y_train = [] # output word
for data_word in data:
    x_train.append(to_one_hot(word2int[ data_word[0] ], vocab_size))
    y_train.append(to_one_hot(word2int[ data_word[1] ], vocab_size))
# convert them to numpy arrays
x_train = np.asarray(x_train)
y_train = np.asarray(y_train)
```

So now we have x_train and y_train:

```
print(x_train)
->
[[ 0.  0.  0.  0.  0.  0.  1.]
 [ 0.  0.  0.  0.  0.  0.  1.]
 [ 0.  0.  0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  1.  0.  0.]
 [ 0.  0.  0.  0.  1.  0.  0.]
 [ 0.  0.  0.  0.  1.  0.  0.]
 [ 0.  0.  0.  1.  0.  0.  0.]
 [ 0.  0.  0.  1.  0.  0.  0.]
 [ 0.  0.  0.  0.  1.  0.  0.]
 [ 0.  0.  0.  0.  1.  0.  0.]
 [ 0.  0.  0.  1.  0.  0.  0.]
 [ 0.  0.  0.  1.  0.  0.  0.]
 [ 0.  0.  0.  1.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  0.  1.  0.]
 [ 0.  1.  0.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.  0.  0.]
 [ 0.  0.  1.  0.  0.  0.  0.]
 [ 0.  0.  1.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  1.  0.  0.]
 [ 0.  0.  0.  0.  1.  0.  0.]
 [ 0.  0.  0.  0.  1.  0.  0.]
 [ 0.  0.  0.  0.  1.  0.  0.]
 [ 0.  1.  0.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.  0.  0.]
 [ 0.  1.  0.  0.  0.  0.  0.]
 [ 1.  0.  0.  0.  0.  0.  0.]
 [ 1.  0.  0.  0.  0.  0.  0.]]
```

Both have the shape:

```
print(x_train.shape, y_train.shape)
->
(34, 7) (34, 7)
# meaning 34 training points, where each point has 7 dimensions
```

Make the tensorflow model

```python
# making placeholders for x_train and y_train
x = tf.placeholder(tf.float32, shape=(None, vocab_size))
y_label = tf.placeholder(tf.float32, shape=(None, vocab_size))
```

Q   △   Upgrade

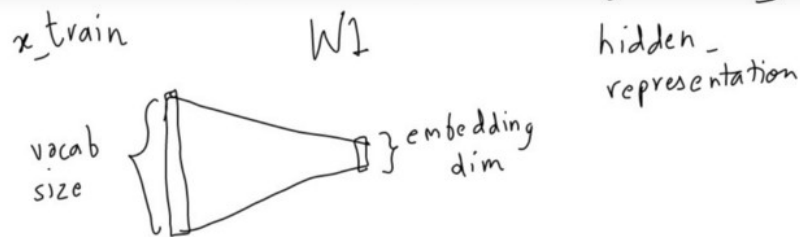DATA SCIENCE   MACHINE LEARNING   PROGRAMMING   VISUALIZATION   AI   JOURNALISM   EVENTS   SUBMIT

Towards Data Science

Sharing concepts, ideas, and codes.

Follow

3.2K

As can be seen in the above diagram, we take our training data and convert into the embedded representation.

```
EMBEDDING_DIM = 5 # you can choose your own number
W1 = tf.Variable(tf.random_normal([vocab_size, EMBEDDING_DIM]))
b1 = tf.Variable(tf.random_normal([EMBEDDING_DIM])) #bias
hidden_representation = tf.add(tf.matmul(x,W1), b1)
```

Next, we take what we have in the embedded dimension and make a prediction about the neighbour. To make the prediction we use softmax.
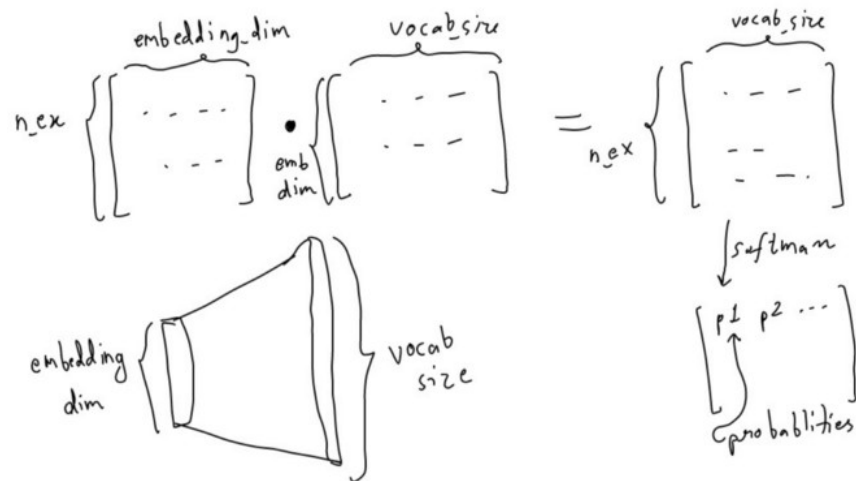

```
W2 = tf.Variable(tf.random_normal([EMBEDDING_DIM, vocab_size]))
b2 = tf.Variable(tf.random_normal([vocab_size]))
prediction = tf.nn.softmax(tf.add( tf.matmul(hidden_representation,
W2), b2))
```

So to summarise:
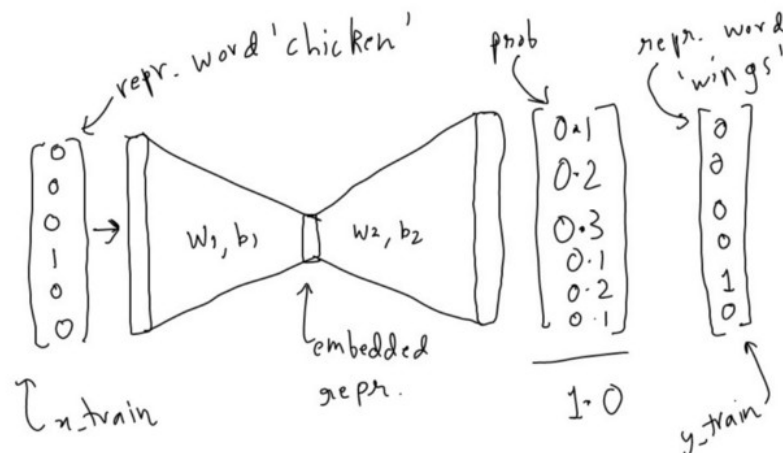

```
input_one_hot  --->  embedded repr. ---> predicted_neighbour_prob
predicted_prob will be compared against a one hot vector to correct
it.
```

Now, all that's left is to train it:

```
sess = tf.Session()
init = tf.global_variables_initializer()
sess.run(init) #make sure you do this!
```

DATA SCIENCE   MACHINE LEARNING   PROGRAMMING   VISUALIZATION   AI   JOURNALISM   EVENTS   SUBMIT

Towards Data Science

Sharing concepts, ideas, and codes.

Follow

🖐 3.2K

🔖

```
# train for n_iter iterations
for _ in range(n_iters):
    sess.run(train_step, feed_dict={x: x_train, y_label: y_train})
    print('loss is : ', sess.run(cross_entropy_loss, feed_dict={x:
x_train, y_label: y_train}))
```

On training, you will get the display of loss:

```
loss is :  2.73213
loss is :  2.30519
loss is :  2.11106
loss is :  1.9916
loss is :  1.90923
loss is :  1.84837
loss is :  1.80133
loss is :  1.76381
loss is :  1.73312
loss is :  1.70745
loss is :  1.68556
loss is :  1.66654
loss is :  1.64975
loss is :  1.63472
loss is :  1.62112
loss is :  1.6087
loss is :  1.59725
loss is :  1.58664
loss is :  1.57676
loss is :  1.56751
loss is :  1.55882
loss is :  1.55064
loss is :  1.54291
loss is :  1.53559
loss is :  1.52865
loss is :  1.52206
loss is :  1.51578
loss is :  1.50979
loss is :  1.50408
loss is :  1.49861
.
.
.
```

It eventually stabilises on a constant loss. Even though we can't get high accuracy, we don't care. All we are interested in is W1 and b1, i.e., the hidden representations.
Let's have a look at them:

```
print(sess.run(W1))
print('----------')
print(sess.run(b1))
print('----------')
->
[[-0.85421133  1.70487809  0.481848   -0.40843448 -0.02236851]
 [-0.47163373  0.34260952 -2.06743765 -1.43854153 -0.14699034]
 [-1.06858993 -1.10739779  0.52600187  0.24079895 -0.46390489]
 [ 0.84426647  0.16476244 -0.72731972 -0.31994426 -0.33553854]
 [ 0.21508843 -1.21030915 -0.13006891 -0.24056002 -0.30445012]
 [ 0.17842589  2.08979321 -0.34172744 -1.8842833  -1.14538431]
 [ 1.61166084 -1.17404735 -0.26805425  0.74437028 -0.81183684]]
----------
[ 0.57727528 -0.83760375  0.19156453 -0.42394346  1.45631313]
----------
```

Why one hot vectors?

$$[0 \quad 0 \quad 0 \quad 1 \quad 0] \times \begin{bmatrix} 17 & 24 & 1 \\ 23 & 5 & 7 \\ 4 & 6 & 13 \\ 10 & 12 & 19 \\ 11 & 18 & 25 \end{bmatrix} = [10 \quad 12 \quad 19]$$

again from Chris McCormick's article (do read it)

When we multiply the one hot vectors with `W1` , we basically get access to the row of the of `W1` which is in fact the embedded representation of the word represented by the input one hot vector.
So `W1` is essentially acting as a look up table.
In our case we have also included a bias term `b1` so you have to add it.

```
vectors = sess.run(W1 + b1)
```

```
[-0.69424796 -1.67628145  3.07313657 -1.14802659 -1.2207377 ]
[-1.7077738  -0.60641652  2.25586247  1.34536338 -0.83848488]
[-0.10080346 -0.90931684  2.8825531  -0.58769202 -1.19922316]
[ 1.49428082 -2.55578995  2.01545811  0.31536022  1.52662396]
[-1.02735448  0.72176981 -0.03772151 -0.60208392  1.53156447]]
```

Towards Data Science

Sharing concepts, ideas, and codes.

Follow

If we want the representation for 'queen', all we have to do is:

```
print(vectors[ word2int['queen'] ])
# say here word2int['queen'] is 2
->
[-0.69424796 -1.67628145  3.07313657 -1.14802659 -1.2207377 ]
```

3.2K

So what can we do with these beautiful beautiful vectors?
Here's a quick function to find the closest vector to a given vector. Beware, it's a dirty
implementation.

```
def euclidean_dist(vec1, vec2):
    return np.sqrt(np.sum((vec1-vec2)**2))
def find_closest(word_index, vectors):
    min_dist = 10000 # to act like positive infinity
    min_index = -1
    query_vector = vectors[word_index]
    for index, vector in enumerate(vectors):
        if euclidean_dist(vector, query_vector) < min_dist and not
np.array_equal(vector, query_vector):
            min_dist = euclidean_dist(vector, query_vector)
            min_index = index
    return min_index
```

We will now query these vectors with 'king', 'queen' and 'royal'

```
print(int2word[find_closest(word2int['king'], vectors)])
print(int2word[find_closest(word2int['queen'], vectors)])
print(int2word[find_closest(word2int['royal'], vectors)])
->
queen
king
he
```

Interesting, our embedding learnt that

```
king is closest to queen
queen is closest to king
royal is closest to he
```

The third one is due to our corpus (It's still pretty good). A bigger corpus will lead to better results.

(*Note: due to the random initialisation of the weights, you might get different results. Run it a few times if required*)
Let's plot them vectors!
First let's reduce the dimensions from 5 to 2 with our favourite dimensionality reduction technique :
tSNE (teesnee!)

```
from sklearn.manifold import TSNE
model = TSNE(n_components=2, random_state=0)
np.set_printoptions(suppress=True)
vectors = model.fit_transform(vectors)
```

Then, we need to normalize the results so that we can view them more comfortably in matplotlib

```
from sklearn import preprocessing
normalizer = preprocessing.Normalizer()
vectors =  normalizer.fit_transform(vectors, 'l2')
```

Finally, we will plot the 2D normalized vectors

```
import matplotlib.pyplot as plt
fig, ax = plt.subplots()
for word in words:
    print(word, vectors[word2int[word]][1])
    ax.annotate(word, (vectors[word2int[word]]
[0],vectors[word2int[word]][1] ))
plt.show()
```

Towards Data Science

Sharing concepts, ideas, and codes.

Follow

👏 3.2K

Wow! `she` is close to `queen` and `king` is equally distanced from `royal` and `queen` We need a bigger corpus to see some of the more complicated relations.

Note: After publishing this, I realized that this example isn't correct because to get meaningful convergence of vectors, we need a really large corpus. The small size of the data makes it vulnerable to sudden "jerks". However, I shall keep this writing intact for pedagogical purposes. For an efficient implementation of word2vec try gensim with some corpus like text8.

Why is this happening?

We have given the neural network the task of predicting neighbours. But we haven't specified how the network should predict it. So, the neural network figures out an *hidden representation* of the words, to aid it in the task of predicting neighbouring words. Predicting neighbouring words is not an interesting task in itself. We care about this hidden representation.

To form these representations, the network is using contexts/neighbours. In our corpus, `king` and `royal` appear as neighbours, and `royal` and `queen` appear as neighbours.

Why predicting neighbours as a task?

Well, other task also manage to form a good representation. Predicting whether the word is a valid n-gram as shown here can also lead to good vectors!

We tried to predict a neighbouring words given a word. This is known as the skip gram model. We could have used neighbouring words of a middle word as inputs and asked the network to predict the middle word. This is known as the Continuous Bag of Words Model.

Further Reading:

This by no means is a complete understanding of word2vec. Part of w2v's beauty is in its 2 modifications to what I've just talked about. These are:

- Negative Sampling
- Hierarchical Softmax

Negative Sampling: it suggests that instead of backpropagating all the 0s in the correct output vector (for a vocab size of 10mill there are 10mill minus 1 zeros) we just backpropagate a few of them (say 14)

Hierarchical Softmax: Calculating the softmax for a vocab of 10mill is very time and computation intensive. Hierarchical Softmax suggests a faster way of computing it using Huffman trees

In the interest of keeping the post simple, I have avoided delving much into it. I definitely recommend reading more into it.

Closing Remarks:

- Word Vectors are super cool
- Don't use this tensorflow code for actual use. It was just to understand. Use a library like gensim

I hope that helped somebody understand these beauties better. If it did, let me know!

If I've made mistakes, **please** let me know.

I would love to connect with you over twitter , linkedin or/and email.

Bye!

Machine Learning   Deep Learning   Word2vec   NLP   TensorFlow

👏 3.2K claps

WRITTEN BY

aneesh joshi

Follow

ML/DL enthusiast | self driving car nanodegree student | ex-game developer | CS undergrad at Manipal Institute of Technology