

Neural Word Embeddings

Blog by Yash R. Shiyani, Intern at MSRF

Some of the most significant recent advances in machine learning and artificial intelligence have come in natural language processing (NLP). Most of the advanced neural architectures in NLP use word embeddings. A word embedding is a representation of a word as a vector of numeric values. For example, the word "night" might be represented as $(-0.076, 0.031, -0.024, 0.022, 0.035)$.

The term "word embedding" doesn't describe the idea very well. Other, less frequently used but more descriptive terms for the same idea include "semantic vector space," "word feature vector" and "vector representation."

The origin of the term "word embedding" is not clear. Several references on the Internet state that the first use of the term was in a 2003 research paper by Y. Bengio et al., but that paper does not use the word "embedding." The earliest usage we were able to find seems to be in a 2005 research paper by F. Morin and Y. Bengio that contains the text, "... a word symbol is first embedded into a Euclidean space."

There are dozens of articles that explain word embeddings, but almost all are either at a very low level of abstraction (all code implementation) or at a very high level (no concrete examples). The goal of this article is to explain word embeddings at a medium level of abstraction, with a bit of code snippets and several examples.

There are three main ways to create a set of word embeddings for an NLP problem. You can create custom word embeddings, or you can use pre-built word embeddings, or you can generate word embeddings on the fly.

Why Are Word Embeddings Needed?

Because neural networks accept only numeric input, if you're working with NLP, words must be converted into numeric values of some sort. Theoretically you could represent words as single integer values, for example, "hello" = 1, "world" = 2, "cat" = 3, and so on, but there are two problems with this approach.

First, because of the way neural networks compute their output, words that are numerically close are indirectly processed as being semantically close. In the example above, "world" and "cat" would be processed as if they were very similar in some way.

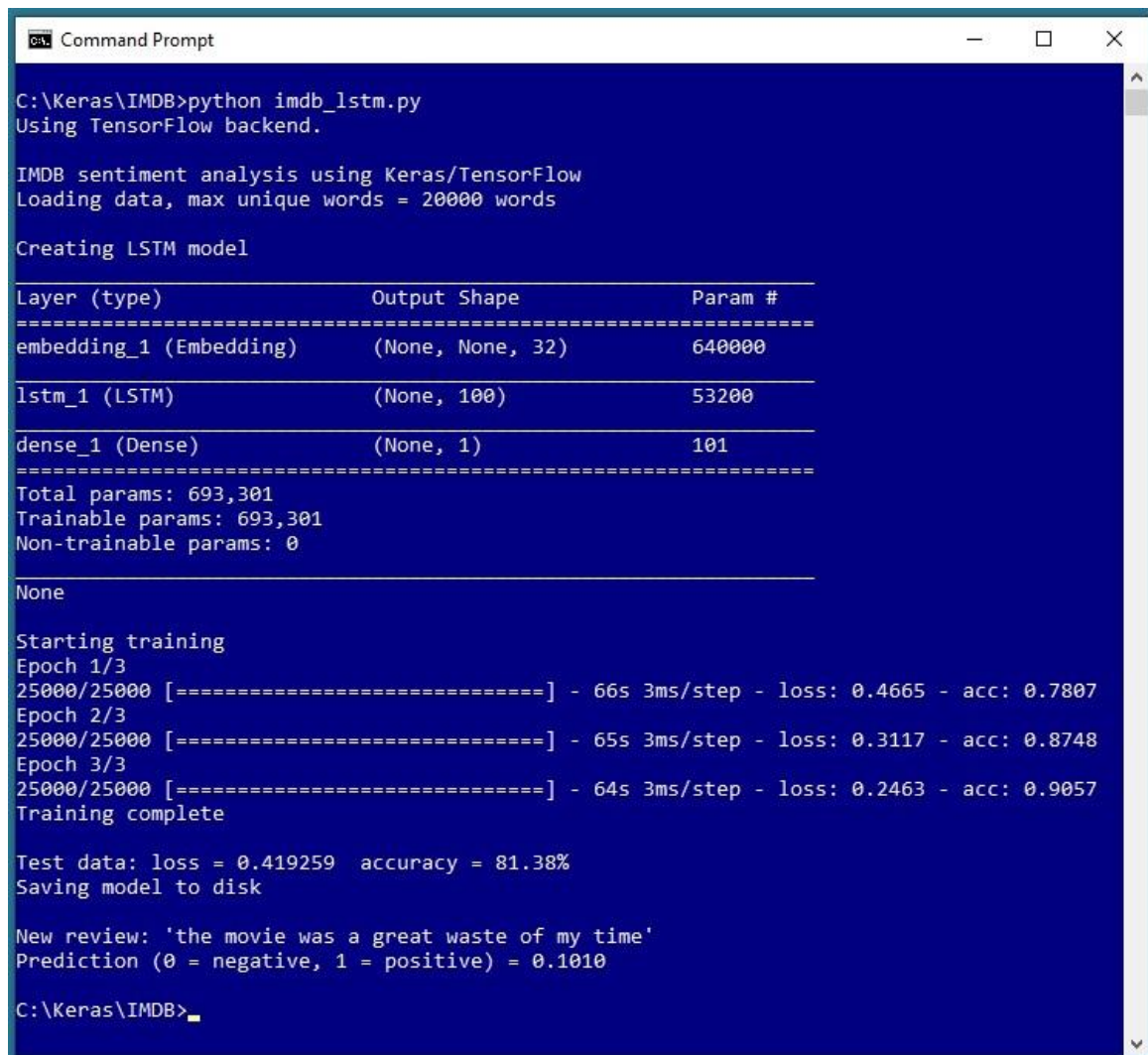
The second problem is that even if you could somehow represent words that are close together in meaning with numeric values that are close, you could only represent similarity in one dimension. For example, the words "man" and "boy" can be compared on the dimension of gender (the same), or on age (a bit different), or on likelihood of being a software engineer (much different).

Creating a Custom Set of Word Embeddings

Although the ideas related to representing words as a numeric vector have been around for decades, arguably the development that led to the current popularity of word embeddings for NLP tasks was the creation and publication of the word2vec algorithm by Google researchers in 2013.

The word2vec algorithm uses a neural approach to map words to numeric vectors. The original word2vec algorithm and tool were patented, but other researchers and engineers quickly created similar algorithms. In some cases these alternate approaches are called word2vec even though technically they're not the same as the original algorithm.

One of the most popular tools for creating a set of custom word embeddings is the gensim library. Gensim is an open source Python library that has many tools for NLP, including a word2vec tool. The connection between gensim, word2vec, and word embeddings is best explained by an example, as shown in **Figure 1**.



```
C:\Keras\IMDB>python imdb_lstm.py
Using TensorFlow backend.

IMDB sentiment analysis using Keras/TensorFlow
Loading data, max unique words = 20000 words

Creating LSTM model
```

Layer (type)	Output Shape	Param #
embedding_1 (Embedding)	(None, None, 32)	640000
lstm_1 (LSTM)	(None, 100)	53200
dense_1 (Dense)	(None, 1)	101

```
=====
Total params: 693,301
Trainable params: 693,301
Non-trainable params: 0
=====
None

Starting training
Epoch 1/3
25000/25000 [=====] - 66s 3ms/step - loss: 0.4665 - acc: 0.7807
Epoch 2/3
25000/25000 [=====] - 65s 3ms/step - loss: 0.3117 - acc: 0.8748
Epoch 3/3
25000/25000 [=====] - 64s 3ms/step - loss: 0.2463 - acc: 0.9057
Training complete

Test data: loss = 0.419259 accuracy = 81.38%
Saving model to disk

New review: 'the movie was a great waste of my time'
Prediction (0 = negative, 1 = positive) = 0.1010

C:\Keras\IMDB>
```

Figure 1: Creating Custom Word Embeddings Using the gensim Library

The demo Python program shown running in Figure 1 begins by setting up a tiny but representation corpus of text, using the first two sentences of a Sherlock Holmes story. The raw text is:

"Mr. Sherlock Holmes, who was usually very late in the mornings, save upon those common occasions when he was up all night, was seated at the breakfast table. I stood upon the fireplace-rug and picked up the stick which our visitor had left behind him the night before."

The demo program hard-codes the corpus like so:

```
mycorpus = [  
    ['mr', 'sherlock', 'holmes', 'who', 'was',  
     'usually', 'very', 'late', 'in', 'the', 'mornings',  
     'save', 'upon', 'those', 'not', 'infrequent',  
     'occasions', 'when', 'he', 'was', 'up', 'all',  
     'night', 'was', 'seated', 'at', 'the', 'breakfast',  
     'table'],  
  
    ['i', 'stood', 'upon', 'the', 'hearth', 'rug', 'and',  
     'picked', 'up', 'the', 'stick', 'which', 'our',  
     'visitor', 'had', 'left', 'behind', 'him', 'the',  
     'night', 'before']  
]
```

The raw text was manually preprocessed by removing all punctuation and converting all characters to lower case. In practice, preprocessing text to create a set of word embeddings is usually very time consuming and is quite tricky. For example, in most situations you would want to retain the single quote character to account for contractions like "it's" (it is) compared to "its" (belonging to). And in some contexts, capitalized words have significance, such as "March" (the month) compared to "march" (walking briskly).

The key lines of code that create and use the custom word embedding are:

```
model = word2vec.Word2Vec(mycorpus, size=5,  
    window=5, min_count=1, sg=0)  
  
print("Embedding vector for 'night' is: ")  
print(model.wv['night'])
```

It's common practice to call a word embedding a model. The size = 5 argument instructs the Word2Vec() method to create a vector with five numeric values for each word in the corpus. In practice, a vector size of 100 or 200 is common.

The window = 5 argument tells Word2Vec() to examine words that are within 5 words of the current word. For example, if window = 2 and the current word being processed

in the Sherlock Holmes corpus is "usually," then the four words "who," "was," (usually), "very" and "late" would be used to determine the word embedding value for "usually."

The `min_count = 1` argument instructs the `Word2Vec()` method to examine all words in the corpus, even rare words that only occur once. In some contexts, you'd want to eliminate rare words. For example, in free form text comments from unknown users, a word that only occurs once could be a misspelling. However, in some situations, a rare word is extremely important, for example "sherlock" in the demo corpus.

The `sg = 0` means "skip-gram equals false". Word embeddings can be created using two different algorithms, skip-gram and continuous bag-of-words (CBOW). The skip-gram algorithm examines each word in the corpus and constructs a neural network that predicts the likelihood of each of the before and after surrounding words. The CBOW algorithm examines a set of words and constructs a neural network that predicts the central word.

In practice, both algorithms tend to give similar results and the best algorithm for a particular problem must be determined by trial and error. In either case, the process of predicting nearby words generates the vectors for each word in the source corpus.

The gensim library `Word2Vec()` method has a total of 25 parameters. Almost all have reasonable default values. The key point is that creating a custom set of word embeddings requires significant effort but gives you maximum flexibility. Creating a custom set of word embeddings is often useful when your problem scenario involves specialized vocabulary and terminology. For example, in a weather context the word "current" would probably refer to an ocean current and be similar to "flow." But in a physics context the word "current" would be similar to "voltage."

The demo program concludes by showing the one word in the corpus that is closest in semantic meaning to "he" using these statements:

```
print("Word most similar to 'he' is: ", end="")
print(model.wv.similar_by_word('he', topn=1)[0][0])
```

The result is the word "holmes," which is quite remarkable because if you examine the raw text you'll see that "he" does in fact refer to "holmes."

Using a Pre-Built Set of Word Embeddings

Several pre-built sets of word embeddings have been created. Two examples are GloVe (global vectors for word representation) and ELMo (embeddings from language models). Both are open source projects. Using a set of pre-built word embeddings is best explained by example.

```
Command Prompt

C:\PureAI\WordEmbeddings>python sentiment_seq_glove.py
Using TensorFlow backend.

Reading reviews into memory

Vocabulary is:
    movie: 1          film: 2
    good: 3           story: 4
    not: 5            bad: 6
    great: 7          nice: 8
    and: 9            plot: 10
    at: 11            all: 12
    excellent: 13     period: 14
    poor: 15          weak: 16
    sad: 17           excuse: 18

Encoded and padded reviews:
[[ 0  0  7  1]
 [ 0  0  8  2]
 [ 3  4  9 10]
 [ 5  6 11 12]
 [ 0  0 13  2]
 [ 0  5  3 14]
 [ 0  0 15  1]
 [ 0  0 16  4]
 [ 0  0  0  6]
 [ 0  0 17 18]]

Loaded 400000 GloVe word vectors.
vec for word 'the' =
( -0.038194 -0.24487 . . . 0.8278 0.27062 )
Starting training
Training complete

Trained model accuracy: 100.00%

Predicting sentiment for phrase 'nice movie'
[[0.  0.  8.  1.]]
Prediction (0 = negative, 1 = positive):
[[0.6857]]

C:\PureAI\WordEmbeddings>
```

Figure 2: Using GloVe Embeddings for Sentiment Analysis

The demo program shown in **Figure 2** performs sentiment analysis using the GloVe pre-built set of word embeddings. The demo begins by setting up 10 tiny movie reviews:

```
reviews = [
    'great movie',
    'nice film',
    'good story and plot',
    'not bad at all',
    'excellent film',

    'not good period',
    'poor movie',
    'weak story',
    'bad',
    'sad excuse']
```

Behind the scenes, the first five reviews are labeled as positive and the second five are labeled as negative. Next, the reviews are programmatically processed, a unique

integer ID is assigned to each word, and then all reviews are padded to a length of four values. The integer-encoded words are:

```
movie: 1    film: 2
good: 3     story: 4
not: 5      bad: 6
great: 7    nice: 8
and: 9      plot: 10
at: 11     all: 12
excellent: 13 period: 14
poor: 15   weak: 16
sad: 17    excuse: 18
```

And so the padded and encoded reviews are:

```
[[ 0  0  7  1]
 [ 0  0  8  2]
 [ 3  4  9 10]
 [ 5  6 11 12]
 [ 0  0 13  2]
 [ 0  5  3 14]
 [ 0  0 15  1]
 [ 0  0 16  4]
 [ 0  0  0  6]
 [ 0  0 17 18]]
```

When working with word embeddings for NLP problems, pre-processing the raw text data is almost always the most difficult and time-consuming part of the project.

The demo program reads the GloVe word embedding into memory:

```
embedding_dict = dict()
file_path = ".\\Data\\glove.6B.100d.txt"

f = open(file_path, "r", encoding="utf8")
for line in f:
    values = line.split()
    word = values[0]
    word_vec = np.asarray(values[1:],
        dtype=np.float32)
    embedding_dict[word] = word_vec
f.close()
```

There are several different versions of GloVe. The demo uses a version that was created from a large corpus of approximately 6 billion words and generated word embedding vectors with 100 values per word. The source corpus is the text of a snapshot of

Wikipedia from 2014, plus an archive of newswire reports from seven different news agencies, from January 2009 through December 2010.

The resulting set of GloVe word embeddings has approximately 400,000 distinct words. The file size is approximately 350 MB which can be easily handled by a desktop PC.

After the GloVe embeddings have been loaded into memory, exactly how to use them depends upon which neural code library is being used. The demo program uses the Keras wrapper library over the TensorFlow neural code library.

The key statements in the demo program that create a simple Keras neural network using the GloVe embeddings are:

```
# define a simple (non-recurrent) model
model = K.models.Sequential()
e = K.layers.Embedding(vocab_size, 100,
weights=[embedding_matrix],
input_length=4, trainable=False)
model.add(e)
model.add(K.layers.Flatten())
model.add(K.layers.Dense(1,
activation='sigmoid'))
```

The key point here is that neural NLP problems are very complex and using word embeddings, either custom-built or pre-built, does not significantly simplify your task. Using word embeddings does, however, often give you much improved NLP models.

Creating Word Embeddings on the Fly

Several neural network code libraries allow you to create word embeddings on the fly. The screenshot in **Figure 3** illustrates the idea.


```
C:\Keras\IMDB>python imdb_lstm.py
Using TensorFlow backend.

IMDB sentiment analysis using Keras/TensorFlow
Loading data, max unique words = 20000 words

Creating LSTM model
```

Layer (type)	Output Shape	Param #
embedding_1 (Embedding)	(None, None, 32)	640000
lstm_1 (LSTM)	(None, 100)	53200
dense_1 (Dense)	(None, 1)	101

```
Total params: 693,301
Trainable params: 693,301
Non-trainable params: 0

None

Starting training
Epoch 1/3
25000/25000 [=====] - 66s 3ms/step - loss: 0.4665 - acc: 0.7807
Epoch 2/3
25000/25000 [=====] - 65s 3ms/step - loss: 0.3117 - acc: 0.8748
Epoch 3/3
25000/25000 [=====] - 64s 3ms/step - loss: 0.2463 - acc: 0.9057
Training complete

Test data: loss = 0.419259 accuracy = 81.38%
Saving model to disk

New review: 'the movie was a great waste of my time'
Prediction (0 = negative, 1 = positive) = 0.1010

C:\Keras\IMDB>
```

Figure 3: Generating Word Embeddings on the Fly with Keras

The demo program uses the Keras code library to create a sentiment analysis model for the well-known IMDB movie reviews dataset. The dataset has 50,000 reviews that are labelled as positive or negative. The dataset is divided into 25,000 reviews intended for training (12,500 positive, 12,500 negative) and 25,000 reviews intended for testing.

Instead of using the training reviews to create a set of custom word embeddings or using a pre-built set of word embeddings, the Keras library allows you to generate custom word embeddings during model training. The key statements that create a sentiment analysis model are:

```
embed_vec_len = 32 # values per word

model = K.models.Sequential()
model.add(K.layers.embeddings.Embedding(input_dim=max_words,
    output_dim=embed_vec_len, embeddings_initializer=e_init,
    mask_zero=True))
model.add(K.layers.LSTM(units=100, kernel_initializer=init,
    dropout=0.2, recurrent_dropout=0.2)) # 100 memory
model.add(K.layers.Dense(units=1, kernel_initializer=init,
    activation='sigmoid'))
model.compile(loss='binary_crossentropy',
    optimizer=simple_adam, metrics=['acc'])
```


The Embedding object is a preliminary layer in front of an LSTM recurrent layer and a Dense standard neural layer. During model training, in addition to finding the values for the weights of the LSTM and Dense layers, the word embeddings for each word in the training set are computed.

The primary advantage of generating word embeddings on the fly is simplicity. You avoid having to deal with the rather messy code to place word embedding values into a neural model. The main disadvantage of generating word embeddings on the fly is that the technique greatly increases training time.

If you examine **Figure 3** you can see that the overall model has 693,301 trainable parameters (weight values that must be computed). Of those values, 640,000 (92 percent) of them are part of the word embeddings and only 53,200 (7 percent) are LSTM weights and just 101 (1 percent) are the standard NN weights. Put another way, approximately 92 percent of the time required to train the model is due to the word embeddings.

In many scenarios, training a neural model involves adjusting model hyperparameters and retraining. Recomputing the word embeddings, which don't change significantly, on every training attempt is very inefficient and time consuming.

Wrapping Up

To summarize, many neural NLP problems require you to convert words into numeric vectors. You can create a set of custom word embeddings using a tool such as gensim. This gives you maximum control over your system at the expense of increased complexity. You can use a set of pre-built word embeddings such as GloVe. This approach is easy but gives you very little control over your word embeddings when you are working with specialized vocabulary and terminology. You can generate word embeddings on the fly. This approach is easy and can handle specialized vocabulary, at the expense of greatly increased time required to train models.

References

<https://wiki.pathmind.com/word2vec>
<https://towardsdatascience.com/introduction-to-word-embedding-and-word2vec-652d0c2060fa>

Github

https://github.com/YashShiyani/Neural_Word_Embeddings