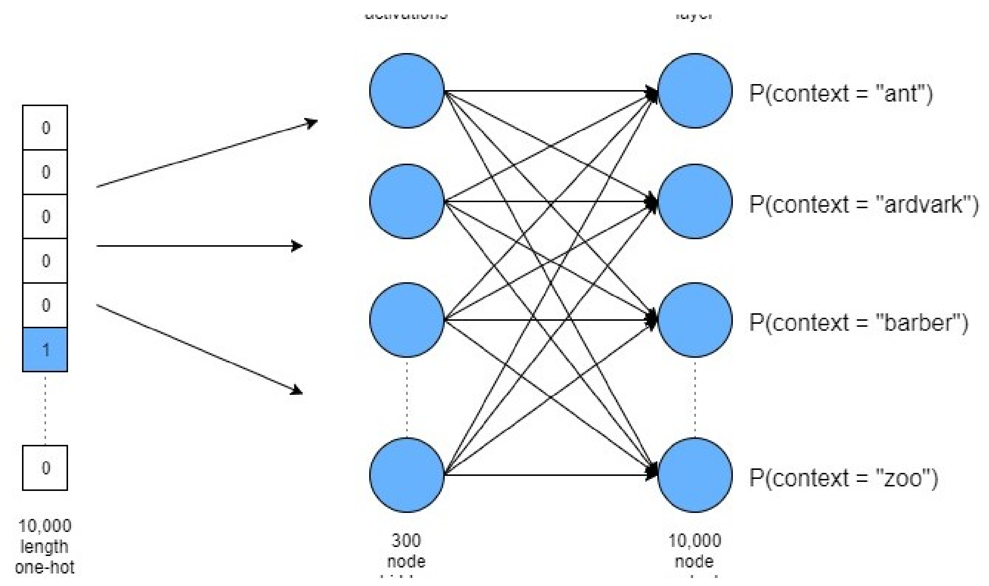
Word2Vec word embedding tutorial in Python and TensorFlow

July 21, 2017 Andy



A Word2Vec softmax trainer

In coming tutorials on this blog I will be dealing with how to create deep learning models that predict text sequences. However, before we get to that point we have to understand some key Natural Language Processing (NLP) ideas. One of the key ideas in NLP is how we can efficiently convert words into numeric vectors which can then be "fed into" various machine learning models to perform predictions. The current key technique to do this is called "Word2Vec" and this is what will be covered in this tutorial. After discussing the relevant background material, we will be implementing Word2Vec embedding using TensorFlow (which makes our lives a lot easier). To get up to speed in TensorFlow, check out my **TensorFlow tutorial**.

Recommended online course: If you are more of a video course learner, check out this inexpensive Udemy course: **Natural Language Processing with Deep Learning in Python**

Why do we need Word2Vec?

If we want to feed words into machine learning models, unless we are using tree based methods, we need to convert the words into some set of numeric vectors. A straight-forward way of doing this would be to use a "one-hot" method of converting the word into a sparse representation with only one element of the vector set to 1, the rest being zero. This is the same method we use for classification tasks – see **this tutorial**.

So, for the sentence "the cat sat on the mat" we would have the following vector representation:

$$\begin{pmatrix} \text{the} \\ \text{cat} \\ \text{sat} \\ \text{on} \\ \text{the} \\ \text{mat} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

(thecatsatonthemat)=(100000100000100000101000000001)

Here we have transformed a six word sentence into a 6×5 matrix, with the 5 being the size of the *vocabulary* ("the" is repeated). In practical applications, however, we will want machine and deep learning models to learn from gigantic vocabularies i.e. 10,000 words plus. You can begin to see the efficiency issue of using "one hot" representations of the words – the input layer into any neural network attempting to model such a vocabulary would have to be at least 10,000 nodes. Not only that, this method strips away any local context of the words – in other words, it strips away information about words which commonly appear close together in sentences (or between sentences).

For instance, we might expect to see "United" and "States" to appear close together, or "Soviet" and "Union". Or "food" and "eat", and so on. This method loses all such information, which, if we are trying to model natural language, is a large omission. Therefore, we need an efficient representation of the text data which also conserves information about local word context. This is where the Word2Vec methodology comes in.

The Word2Vec methodology

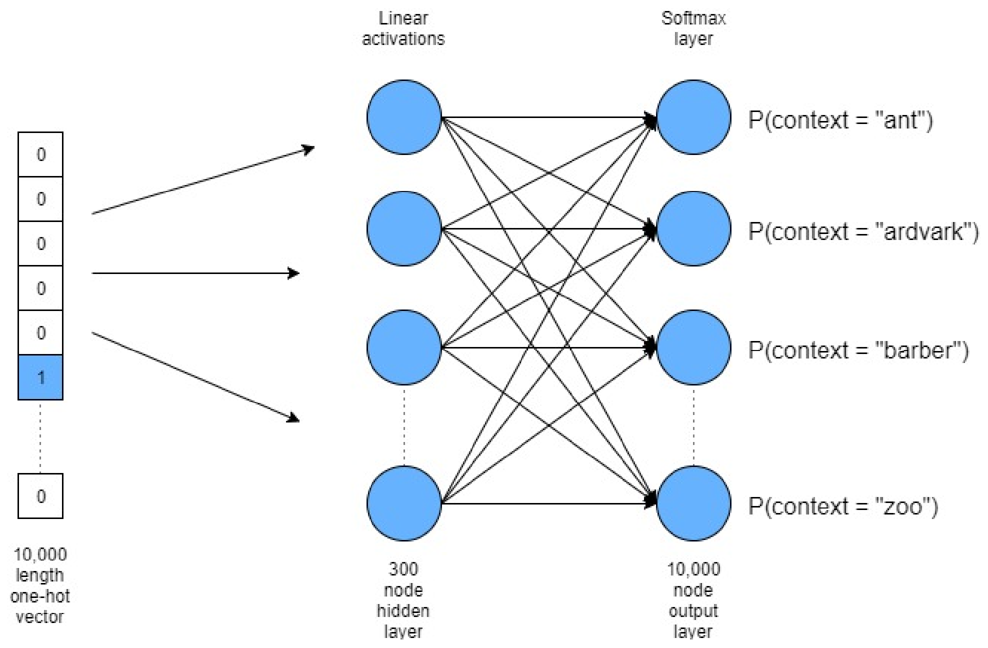
As mentioned previously, there is two components to the Word2Vec methodology. The first is the mapping of a high dimensional one-hot style representation of words to a lower dimensional vector. This might involve transforming a 10,000 columned matrix into a 300 columned matrix, for instance. This process is called word embedding. The second goal is to do this while still maintaining word context and therefore, to some extent, meaning. One approach to achieving these two goals in the Word2Vec methodology is by taking an input word and then attempting to estimate the probability of other words appearing close to that word. This is called the skip-gram approach. The alternative method, called Continuous Bag Of Words (CBOW), does the opposite – it takes some context words as input and tries to find the single word that has the highest probability of fitting that context. In this tutorial, we will concentrate on the skip-gram method.

What's a gram? A gram is a group of *n* words, where *n* is the gram window size. So for the sentence "The cat sat on the mat", a 3-gram representation of this sentence would be "The cat sat", "cat sat on", "sat on the", "on the mat". The "skip" part refers to the number of times an input word is repeated in the data-set with different context words (more on this later). These grams are fed into the Word2Vec context prediction system. For instance, assume the input word is "cat" – the Word2Vec tries to predict the context ("the", "sat") from this supplied input word. The Word2Vec system will move through all the supplied grams and input words and attempt to learn appropriate mapping vectors (embeddings) which produce high probabilities for the right context given the input words.

What is this Word2Vec prediction system? Nothing other than a neural network.

The softmax Word2Vec method

Consider the diagram below – in this case we'll assume the sentence "The cat sat on the mat" is part of a much larger text database, with a very large vocabulary – say 10,000 words in length. We want to reduce this to a 300 length embedding.



A Word2Vec softmax trainer

With respect to the diagram above, if we take the word "cat" it will be one of the words in the 10,000 word vocabulary. Therefore we can represent it as a 10,000 length one-hot vector. We then interface this input vector to a 300 node hidden layer (if you need to scrub up on neural networks, see **this tutorial**). The weights connecting this layer will be our new word vectors – more on this soon. The activations of the nodes in this hidden layer are simply linear summations of the weighted inputs (i.e. no non-linear activation, like a sigmoid or tanh, is applied). These nodes are then fed into a softmax output layer. During training, we want to change the weights of this neural network so that words surrounding "cat" have a higher probability in the softmax output layer. So, for instance, if our text data set has a lot of Dr Seuss books, we would want our network to assign large probabilities to words like "the", "sat" and "on" (given lots of sentences like "the cat sat on the mat").

By training this network, we would be creating a 10,000 x 300 weight matrix connecting the 10,000 length input with the 300 node hidden layer. Each row in this matrix corresponds to a word in our 10,000 word vocabulary – so we have effectively reduced 10,000 length one-hot vector representations of our words to 300 length vectors. The weight matrix essentially becomes a look-up or encoding table of our words. Not only that, but these weight values contain context information due to the way we've trained our network. Once we've trained the network, we abandon the softmax layer and just use the 10,000 x 300 weight matrix as our word embedding lookup table.

What does this look like in code?

The softmax Word2Vec method in TensorFlow

As with any machine learning problem, there are two components – the first is getting all the data into a usable format, and the next is actually performing the training, validation and testing. First I'll go through how the data can be gathered into a usable format, then we'll talk about the TensorFlow graph of the model. Note that the code that I will be going through can be found in its entirety at this site's **Github repository**. In this case, the code is mostly based on the TensorFlow Word2Vec tutorial **here** with some personal changes.

Preparing the text data

The previously mentioned TensorFlow tutorial has a few functions that take a text database and transform it so that we can extract input words and their associated grams in mini-batches for training the Word2Vec system / embeddings (if you're not sure what "mini-batch" means, check out **this tutorial**). I'll briefly talk about each of these functions in turn:

```
def maybe_download(filename, url, expected_bytes):
    """Download a file if not present, and make sure it's the right size."""
    if not os.path.exists(filename):
        filename, _ = urllib.request.urlretrieve(url + filename, filename)
    statinfo = os.stat(filename)
    if statinfo.st_size == expected_bytes:
        print('Found and verified', filename)
    else:
        print(statinfo.st_size)
        raise Exception(
            'Failed to verify ' + filename + '. Can you get to it with a browser?')
    return filename
```

This function checks to see if the *filename* already has been downloaded from the supplied *url*. If not, it uses the urllib.request Python module which retrieves a file from the given *url* argument, and downloads the file into the local code directory. If the file already exists (i.e. *os.path.exists(filename)*returns true), then the function does not try to download the file again. Next, the function checks the size of the file and makes sure it lines up with the expected file size, *expected_bytes*. If all is well, it returns the *filename* object which can be used to extract the data from. To call the function with the data-set we are using in this example, we execute the following code:

```
url = 'http://mattmahoney.net/dc/'
filename = maybe_download('text8.zip', url, 31344016)
```

The next thing we have to do is take the *filename* object, which points to the downloaded file, and extract the data using the Python *zipfile* module.

```
# Read the data into a list of strings.
def read_data(filename):
    """Extract the first file enclosed in a zip file as a list of words."""
    with zipfile.ZipFile(filename) as f:
        data = tf.compat.as_str(f.read(f.namelist()[0])).split()
    return data
```

Using *zipfile.ZipFile()* to extract the zipped file, we can then use the reader functionality found in this *zipfile* module. First, the *namelist()* function retrieves all the members of the archive – in this case there is only one member, so we access this using the zero index. Then we use the *read()*function which reads all the text in the file and pass this through the TensorFlow function *as_str* which ensures that the text is created as a string data-type. Finally, we use *split()* function to create a list with all the words in the text file, separated by white-space characters. We can see some of the output here:

```
vocabulary = read_data(filename)
print(vocabulary[:7])
['anarchism', 'originated', 'as', 'a', 'term', 'of', 'abuse']
```

As you can observe, the returned vocabulary data contains a list of plain English words, ordered as they are in the sentences of the original extracted text file. Now that we have all the words extracted in a list, we have to do some further processing to enable us to create our skip-gram batch data. These further steps are:

1. Extract the top 10,000 most common words to include in our embedding vector
2. Gather together all the unique words and index them with a unique integer value – this is what is required to create an equivalent one-hot type input for the word. We'll use a dictionary to do this

3. Loop through every word in the dataset (*vocabulary* variable) and assign it to the unique integer word identified, created in Step 2 above. This will allow easy lookup / processing of the word data stream

The function which performs all this magic is shown below:

```
def build_dataset(words, n_words):
    """Process raw inputs into a dataset."""
    count = [['UNK', -1]]
    count.extend(collections.Counter(words).most_common(n_words - 1))
    dictionary = dict()
    for word, _ in count:
        dictionary[word] = len(dictionary)
    data = list()
    unk_count = 0
    for word in words:
        if word in dictionary:
            index = dictionary[word]
        else:
            index = 0  # dictionary['UNK']
            unk_count += 1
        data.append(index)
    count[0][1] = unk_count
    reversed_dictionary = dict(zip(dictionary.values(), dictionary.keys()))
    return data, count, dictionary, reversed_dictionary
```

The first step is setting up a “counter” list, which will store the number of times a word is found within the data-set. Because we are restricting our vocabulary to only 10,000 words, any words not within the top 10,000 most common words will be marked with an “UNK” designation, standing for “unknown”. The initialized *count* list is then extended, using the Python collections module and the *Counter()* class and the associated *most_common()* function. These count the number of words in the given argument (*words*) and then returns the *n* most common words in a list format.

The next part of this function creates a dictionary, called *dictionary* which is populated by keys corresponding to each unique word. The value assigned to each unique word key is simply an increasing integer count of the size of the dictionary. So, for instance, the most common word will receive the value 1, the second most common the value 2, the third most common word the value 3, and so on (the integer 0 is assigned to the ‘UNK’ words).   This step creates a unique integer value for each word within the vocabulary – accomplishing the second step of the process which was defined above.

Next, the function loops through each *word* in our full *words* data set – the data set which was output from the read_data() function. A list called *data* is created, which will be the same length as *words* but instead of being a list of individual words, it will instead be a list of integers – with each word now being represented by the unique integer that was assigned to this word in *dictionary.* So, for the first sentence of our data-set [‘anarchism’, ‘originated’, ‘as’, ‘a’, ‘term’, ‘of’, ‘abuse’], now looks like this in the *data*variable: [5242, 3083, 12, 6, 195, 2, 3136]. This part of the function addresses step 3 in the list above.

Finally, the function creates a dictionary called *reverse_dictionary* that allows us to look up a word based on its unique integer identifier, rather than looking up the identifier based on the word i.e. the original *dictionary.*

The final aspect of setting up our data is now to create a data set comprising of our input words and associated grams, which can be used to train our Word2Vec embedding system. The code to do this is:

```
data_index = 0
# generate batch data
def generate_batch(data, batch_size, num_skips, skip_window):
    global data_index
    assert batch_size % num_skips == 0
    assert num_skips <= 2 * skip_window
    batch = np.ndarray(shape=(batch_size), dtype=np.int32)
    context = np.ndarray(shape=(batch_size, 1), dtype=np.int32)
    span = 2 * skip_window + 1  # [ skip_window input_word skip_window ]
    buffer = collections.deque(maxlen=span)
    for _ in range(span):
        buffer.append(data[data_index])
        data_index = (data_index + 1) % len(data)
    for i in range(batch_size // num_skips):
        target = skip_window  # input word at the center of the buffer
        targets_to_avoid = [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] = buffer[skip_window]  # this is the input word
            context[i * num_skips + j, 0] = buffer[target]  # these are the context words
        buffer.append(data[data_index])
        data_index = (data_index + 1) % len(data)
    # Backtrack a little bit to avoid skipping words in the end of a batch
    data_index = (data_index + len(data) - span) % len(data)
    return batch, context
```

This function will generate mini-batches to use during our training (again, see **here** for information on mini-batch training). These batches will consist of input words (stored in *batch*) and random associated context words within the gram as the labels to predict (stored in *context*). For instance, in the 5-gram "the cat sat on the", the input word will be center word i.e. "sat" and the context words that will be predicted will be drawn randomly from the remaining words of the gram: ['the', 'cat', 'on', 'the']. In this function, the number of words drawn randomly from the surrounding context is defined by the argument *num_skips*. The size of the window of context words to draw from around the input word is defined in the argument *skip_window* – in the example above ("the cat sat on the"), we have a skip window width of 2 around the input word "sat".

In the function above, first the batch and label outputs are defined as variables of size *batch_size*. Then the span size is defined, which is basically the size of the word list that the input word and context samples will be drawn from. In the example sub-sentence above "the cat sat on the", the span is 5 = 2 x skip window + 1. After this a buffer is created:

```
buffer = collections.deque(maxlen=span)
for _ in range(span):
    buffer.append(data[data_index])
    data_index = (data_index + 1) % len(data)
```

This buffer will hold a maximum of *span* elements and will be a kind of moving window of words that samples are drawn from. Whenever a new word index is added to the buffer, the left most element will drop out of the buffer to allow room for the new word index being added. The position of the buffer in the input text stream is stored in a global variable *data_index*which is incremented each time a new word is added to the buffer. If it gets to the end of the text stream, the "% len(data)" component of the index update will basically reset the count back to zero.

The code below fills out the batch and context variables:

```
for i in range(batch_size // num_skips):
    target = skip_window  # input word at the center of the buffer
    targets_to_avoid = [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] = buffer[skip_window]  # this is the input word
    context[i * num_skips + j, 0] = buffer[target]  # these are the context words
buffer.append(data[data_index])
data_index = (data_index + 1) % len(data)
```

The first "target" word selected is the word at the center of the span of words and is therefore the input word. Then other words are randomly selected from the span of words, making sure that the input word is not selected as part of the context, and each context word is unique. The *batch*variable will feature repeated input words (*buffer[skip_window]*) which are matched with each context word in *context*.

The *batch* and *context* variables are then returned – and now we have a means of drawing batches of data from the data set. We are now in a position to create our Word2Vec training code in TensorFlow. However, before we get to that, we'll first create a validation data-set that we can use to test how our model is doing. We do that by measuring the vectors closest together in vector-space, and make sure these words indeed are similar using our knowledge of English. This will be discussed more in the next section. However, for now, the code below shows how to grab some random validation words from the most common words in our vocabulary:

```
# We pick a random validation set to sample nearest neighbors. Here we limit the
# validation samples to the words that have a low numeric ID, which by
# construction are also the most frequent.
valid_size = 16     # Random set of words to evaluate similarity on.
valid_window = 100  # Only pick dev samples in the head of the distribution.
valid_examples = np.random.choice(valid_window, valid_size, replace=False)
```

The code above randomly chooses 16 integers from 0-100 – this corresponds to the integer indexes of the most common 100 words in our text data. These will be the words we examine to assess how our learning is progressing in associating related words together in the vector-space. Now, onto creating the TensorFlow model.

Creating the TensorFlow model

For a refresher on TensorFlow, check out **this tutorial**. Below I will step through the process of creating our Word2Vec word embeddings in TensorFlow. What does this involve? Simply, we need to setup the neural network which I previously presented, with a word embedding matrix acting as the hidden layer and an output softmax layer in TensorFlow. By training this model, we'll be learning the best word embedding matrix and therefore we'll be learning a reduced, context maintaining, mapping of words to vectors.

The first thing to do is set-up some variables which we'll use later on in the code – the purposes of these variables will become clear as we progress:

```
batch_size = 128
embedding_size = 128  # Dimension of the embedding vector.
skip_window = 1       # How many words to consider left and right.
num_skips = 2         # How many times to reuse an input to generate a context.
```

Next we setup some TensorFlow placeholders that will hold our input words (their integer indexes) and context words which we are trying to predict. We also need to create a constant to hold our validation set indexes in TensorFlow:

```
train_inputs = tf.placeholder(tf.int32, shape=[batch_size])
train_labels = tf.placeholder(tf.int32, shape=[batch_size, 1])
valid_dataset = tf.constant(valid_examples, dtype=tf.int32)
```

Next, we need to setup the embedding matrix variable / tensor – this is straight-forward using the TensorFlow *embedding_lookup()* function, which I'll explain shortly:

```
# Look up embeddings for inputs.
embeddings = tf.Variable(
    tf.random_uniform([vocabulary_size, embedding_size], -1.0, 1.0))
embed = tf.nn.embedding_lookup(embeddings, train_inputs)
```

The first step in the code above is to create the embeddings variable, which is effectively the weights of the connections to the linear hidden layer. We initialize the variable with a random uniform distribution between -1.0 to 1.0. The size of this variable is (*vocabulary_size, embedding_size*) – the *vocabulary_size* is the 10,000 words that we have used to setup our data in the previous section. This is basically our one-hot vector input, where the only element with a value of "1" is the current input word, all the other values are set to "0". The second dimension, *embedding_size,* is our hidden layer size, and is the length of our new, smaller, representation of our words. We can also think of this tensor as a big lookup table – the rows are each word in our vocabulary, and the columns are our new vector representation of each of these words. Here's a simplified example (using dummy values), where *vocabulary_size=7* and *embedding_size=3*:

$$\begin{array}{c|ccc}
\text{anarchism} & 0.5 & 0.1 & -0.1 \\
\text{originated} & -0.5 & 0.3 & 0.9 \\
\text{as} & 0.3 & -0.5 & -0.3 \\
\text{a} & 0.7 & 0.2 & -0.3 \\
\text{term} & 0.8 & 0.1 & -0.1 \\
\text{of} & 0.4 & -0.6 & -0.1 \\
\text{abuse} & 0.7 & 0.1 & -0.4
\end{array}$$

anarchism0.50.1−0.1originated−0.50.30.9as0.3−0.5−0.3a0.70.2−0.3term0.80.1−0.1of0.4−0.6−0.1abuse0.70.1−0.4

As can be observed, "anarchism" (which would actually be represented by a unique integer or one-hot vector) is now expressed as [0.5, 0.1, -0.1]. We can "look up" anarchism by finding its integer index and searching the rows of *embeddings* to find the embedding vector: [0.5, 0.1, -0.1].

The next line in the code involves the *tf.nn.embedding_lookup()* function, which is a useful helper function in TensorFlow for this type of task. Here's how it works – it takes an input vector of integer indexes – in this case our *train_input* tensor of training input words, and "looks up" these indexes in the supplied embeddings tensor.

Therefore, this command will return the current embedding vector for each of the supplied input words in the training batch. The full embedding tensor will be optimized during the training process.

Next we have to create some weights and bias values to connect the output softmax layer, and perform the appropriate multiplication and addition. This looks like:

```
# Construct the variables for the softmax
weights = tf.Variable(tf.truncated_normal([vocabulary_size, embedding_size],
                          stddev=1.0 / math.sqrt(embedding_size)))
biases = tf.Variable(tf.zeros([vocabulary_size]))
hidden_out = tf.matmul(embed, tf.transpose(weights)) + biases
```

The weight variable, as it is connecting the hidden layer and the output layer, is of size (*out_layer_size, hidden_layer_size) = (vocabulary_size, embedding_size)*. The biases, as usual, will only be single dimensional and the size of the output layer. We then multiply the embedded variable (*embed*) by the weights and add the bias. Now we are ready to create a softmax operation and we will use cross entropy loss to optimize the weights, biases and embeddings of the model. To do this easily, we will use the TensorFlow function *softmax_cross_entropy_with_logits()*. However, to use this function we first have to convert the context words / integer indices into one-hot vectors. The code below performs both of these steps, and also adds a gradient descent optimization operation:

```
# convert train_context to a one-hot format
train_one_hot = tf.one_hot(train_context, vocabulary_size)
cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=hidden_out,
    labels=train_one_hot))
# Construct the SGD optimizer using a learning rate of 1.0.
optimizer = tf.train.GradientDescentOptimizer(1.0).minimize(cross_entropy)
```

Next, we need to perform our similarity assessments to check on how the model is performing as it trains. To determine which words are similar to each other, we need to perform some sort of operation that measures the "distances" between the various word embedding vectors for the different words. In this case, we will use the **cosine similarity** measure of distance between vectors. It is defined as:

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|_2 \|\mathbf{B}\|_2}$$

similarity=cos(θ)=A·B‖A‖2‖B‖2

Here the bolded ***A*** and ***B*** are the two vectors that we are measuring the similarity between. The double parallel lines with the *2* subscript ($\|A\|_2$ ‖A‖2) refers to the L2 norm of the vector. To get the L2 norm of a vector, you square every dimension of the vector (in this case *n=300,* the width of our embedding vector), sum up the squared elements then take the square root of the product i.e.:

$$\sqrt{\sum_{i=1}^{n} A_i^2}$$

∑i=1nAi2

The best way to calculate the cosine similarity in TensorFlow is to normalize each vector like so:

$$\frac{A}{\|A\|_2}$$

A‖A‖2

Then we can simply multiply these normalized vectors together to get the cosine similarity. We will multiply the validation vectors/words that were discussed earlier with all of the words in our embedding vector, then we can sort in descending order to get those words most similar to our validation words.

First, we calculate the L2 norm of each vector using the *tf.square()*, *tf.reduce_sum()* and *tf.sqrt()* functions to calculate the square, summation and square root of the norm, respectively:

```
# Compute the cosine similarity between minibatch examples and all embeddings.
norm = tf.sqrt(tf.reduce_sum(tf.square(embeddings), 1, keep_dims=True))
normalized_embeddings = embeddings / norm
```

Now we can look up our validation words / vectors using the *tf.nn.embedding_lookup()* that we discussed earlier:

```
valid_embeddings = tf.nn.embedding_lookup(
      normalized_embeddings, valid_dataset)
```

As before, we are supplying a list of integers (that correspond to our validation vocabulary words) to the *embedding_lookup()* function, which looks up these rows in the *normalized_embeddings* tensor, and returns the subset of *validation* normalized embeddings. Now that we have the normalized validation tensor, *valid_embeddings*, we can multiply this by the full normalized vocabulary (*normalized_embedding*) to finalize our similarity calculation:

```
similarity = tf.matmul(
      valid_embeddings, normalized_embeddings, transpose_b=True)
```

This operation will return a (*validation_size, vocabulary_size*) sized tensor, where each row refers to one of our validation words and the columns refer to the similarity between the validation word and all the other words in the vocabulary.

Running the TensorFlow model

The code below initializes the variables and feeds in each data batch to the training loop, printing the average loss every 2000 iterations. If this code doesn't make sense to you, check out my **TensorFlow tutorial**.

```
with tf.Session(graph=graph) as session:
  # We must initialize all variables before we use them.
  init.run()
  print('Initialized')

  average_loss = 0
  for step in range(num_steps):
    batch_inputs, batch_context = generate_batch(data,
        batch_size, num_skips, skip_window)
    feed_dict = {train_inputs: batch_inputs, train_context: batch_context}

    # We perform one update step by evaluating the optimizer op (including it
    # in the list of returned values for session.run()
    _, loss_val = session.run([optimizer, cross_entropy], feed_dict=feed_dict)
    average_loss += loss_val

    if step % 2000 == 0:
      if step > 0:
        average_loss /= 2000
      # The average loss is an estimate of the loss over the last 2000 batches.
      print('Average loss at step ', step, ': ', average_loss)
      average_loss = 0
```

Next, we want to print out the words which are most similar to our validation words – we do this by calling the similarity operation we defined above and sorting the results (note, this is only performed every 10,000 iterations as it is computationally expensive):

```
# Note that this is expensive (~20% slowdown if computed every 500 steps)
if step % 10000 == 0:
    sim = similarity.eval()
    for i in range(valid_size):
        valid_word = reverse_dictionary[valid_examples[i]]
        top_k = 8  # number of nearest neighbors
        nearest = (-sim[i, :]).argsort()[1:top_k + 1]
        log_str = 'Nearest to %s:' % valid_word
        for k in range(top_k):
            close_word = reverse_dictionary[nearest[k]]
            log_str = '%s %s,' % (log_str, close_word)
        print(log_str)
```

This function first evaluates the similarity operation, which returns an array of cosine similarity values for each of the validation words. Then we iterate through each of the validation words, taking the top 8 closest words by using argsort() on the negative of the similarity to arrange the values in descending order. The code then prints out these 8 closest words so we can monitor how the embedding process is performing.

Finally, after all the training iterations are finished, we can assign the final embeddings to a separate tensor for use later (most likely in some sort of other deep learning or machine learning process):

```
final_embeddings = normalized_embeddings.eval()
```

So now we're done – or are we? The code for this softmax method of Word2Vec is on this site's Github repository – you could try running it, but I wouldn't recommend it. Why? Because it is seriously slow.

Speeding things up – the "true" Word2Vec method

The fact is, performing softmax evaluations and updating the weights over a 10,000 word output/vocabulary is really slow. Why's that? Consider the softmax definition:

$$P(y=j \mid x) = \frac{e^{x^T w_j}}{\sum_{k=1}^{K} e^{x^T w_k}}$$

P(y=j|x)=exTwj∑k=1KexTwk

In the context of what we are working on, the softmax function will predict what words have the highest probability of being in the context of the input word. To determine that probability however, the denominator of the softmax function has to evaluate *all* the possible context words in the vocabulary. Therefore, we need 300 x 10,000 = 3M weights, all of which need to be trained for the softmax output. This slows things down.

There is an alternative, faster scheme called **Noise Contrastive Estimation (NCE)**. Instead of taking the probability of the context word compared to *all* of the possible context words in the vocabulary, this method randomly samples 2-20 possible context words and evaluates the probability only from these. I won't go into the nitty gritty details here, but suffice to say that this method has been shown to perform well and drastically speeds up the training process.

TensorFlow has helped us out here, and has supplied an NCE loss function that we can use called *tf.nn.nce_loss()* which we can supply weight and bias variables to. Using this function, the time to perform 100 training iterations reduced from 25 seconds with the softmax method to less than 1 second using the NCE method. An awesome improvement! We replace the softmax lines with the following in our code:

```
# Construct the variables for the NCE loss
nce_weights = tf.Variable(
        tf.truncated_normal([vocabulary_size, embedding_size],
                            stddev=1.0 / math.sqrt(embedding_size)))
nce_biases = tf.Variable(tf.zeros([vocabulary_size]))

nce_loss = tf.reduce_mean(
        tf.nn.nce_loss(weights=nce_weights,
                       biases=nce_biases,
                       labels=train_context,
                       inputs=embed,
                       num_sampled=num_sampled,
                       num_classes=vocabulary_size))

optimizer = tf.train.GradientDescentOptimizer(1.0).minimize(nce_loss)
```

Now we are good to run the code. You can get the full code **here**. As discussed, every 10,000 iterations the code outputs the validation words and the words that the Word2Vec system deems are similar. Below, you can see the improvement for some selected validation words between the random initialization and at the 50,000 iteration mark:

At the beginning:

> *Nearest to nine: heterosexual, scholarly, scandal, serves, humor, realized, cave, himself*
>
> *Nearest to this: contains, alter, numerous, harmonica, nickname, ghana, bogart, marxist*

After 10,000 iterations:

> *Nearest to nine: zero, one, and, coke, in, UNK, the, jpg*
>
> *Nearest to this: the, a, UNK, killing, meter, afghanistan, ada, indiana*

Finally after 50,000 iterations:

> *Nearest to nine: eight, one, zero, seven, six, two, five, three*
>
> *Nearest to this: that, the, a, UNK, one, it, he, an*

By examining the outputs above, we can first see that the word “nine” becomes increasingly associated with other number words (“eight”, “one”, “seven” etc.). This makes sense. The word “this”, which acts as a pronoun and definite article in sentences, becomes associated with other pronouns (“he”, “it”) and other definite articles (“the”, “that”, etc.) the more iterations we run.

In summary then, we have learnt how to use the Word2Vec methodology to reduce large one-hot word vectors to much reduced word embedding vectors which preserve the context and meaning of the original words. These word embedding vectors can then be used as a more efficient and effective input to deep learning techniques which aim to model natural language. These techniques, such as recurrent neural networks, will be the subject of future posts.

Recommended online course: If you would like to learn more in a video format, check out this well rated and inexpensive Udemy course: **Natural Language Processing with Deep Learning in Python**