Word2Vec Tutorial - The Skip-Gram Model
19 Apr 2016

This tutorial covers the skip gram neural network architecture for Word2Vec. My intention with this tutorial was to skip over the usual introductory and abstract insights about Word2Vec, and get into more of the details. Specifically here I’m diving into the skip gram neural network model.

The Model

The skip-gram neural network model is actually surprisingly simple in its most basic form; I think it’s all of the little tweaks and enhancements that start to clutter the explanation.

Let’s start with a high-level insight about where we’re going. Word2Vec uses a trick you may have seen elsewhere in machine learning. We’re going to train a simple neural network with a single hidden layer to perform a certain task, but then we’re not actually going to use that neural network for the task we trained it on! Instead, the goal is actually just to learn the weights of the hidden layer--we’ll see that these weights are actually the “word vectors” that we’re trying to learn.

Another place you may have seen this trick is in unsupervised feature learning, where you train an auto-encoder to compress an input vector in the hidden layer, and decompress it back to the original in the output layer. After training it, you strip off the output layer (the decompression step) and just use the hidden layer--it’s a trick for learning good image features without having labeled training data.

The Fake Task

So now we need to talk about this “fake” task that we’re going to build the neural network to perform, and then we’ll come back later to how this indirectly gives us those word vectors that we are really after.

We’re going to train the neural network to do the following. Given a specific word in the middle of a sentence (the input word), look at the words nearby and pick one at random. The network is going to tell us the probability for every word in our vocabulary of being the “nearby word” that we chose.
When I say "nearby", there is actually a "window size" parameter to the algorithm. A typical window size might be 5, meaning 5 words behind and 5 words ahead (10 in total).

The output probabilities are going to relate to how likely it is find each vocabulary word nearby our input word. For example, if you gave the trained network the input word “Soviet”, the output probabilities are going to be much higher for words like “Union” and “Russia” than for unrelated words like “watermelon” and “kangaroo”.

We’ll train the neural network to do this by feeding it word pairs found in our training documents. The below example shows some of the training samples (word pairs) we would take from the sentence “The quick brown fox jumps over the lazy dog.” I’ve used a small window size of 2 just for the example. The word highlighted in blue is the input word.

### Source Text

- The **quick** brown fox jumps over the lazy dog.
- The **quick** brown **fox** jumps over the lazy dog.
- The **quick** **brown** fox jumps **over** the lazy dog.
- The **quick** **brown** fox **jumps** over the lazy dog.

### Training Samples

- (the, quick)
- (the, brown)
- (quick, the)
- (quick, brown)
- (quick, fox)
- (brown, the)
- (brown, quick)
- (brown, fox)
- (brown, jumps)
- (fox, quick)
- (fox, brown)
- (fox, jumps)
- (fox, over)

The network is going to learn the statistics from the number of times each pairing shows up. So, for example, the network is probably going to get many more training samples of ("Soviet", "Union") than it is of ("Soviet", "Sasquatch"). When the training is finished, if you give it the word “Soviet” as input, then it will output a much higher probability for “Union” or “Russia” than it will for “Sasquatch”.

## Model Details

So how is this all represented?

First of all, you know you can’t feed a word just as a text string to a neural network, so we need
a way to represent the words to the network. To do this, we first build a vocabulary of words from our training documents—let’s say we have a vocabulary of 10,000 unique words.

We’re going to represent an input word like “ants” as a one-hot vector. This vector will have 10,000 components (one for every word in our vocabulary) and we’ll place a “1” in the position corresponding to the word “ants”, and 0s in all of the other positions.

The output of the network is a single vector (also with 10,000 components) containing, for every word in our vocabulary, the probability that a randomly selected nearby word is that vocabulary word.

Here’s the architecture of our neural network.

There is no activation function on the hidden layer neurons, but the output neurons use softmax. We’ll come back to this later.

When training this network on word pairs, the input is a one-hot vector representing the input word and the training output is also a one-hot vector representing the output word. But when you evaluate the trained network on an input word, the output vector will actually be a probability distribution (i.e., a bunch of floating point values, not a one-hot vector).

The Hidden Layer

For our example, we’re going to say that we’re learning word vectors with 300 features. So the hidden layer is going to be represented by a weight matrix with 10,000 rows (one for every
word in our vocabulary) and 300 columns (one for every hidden neuron).

300 features is what Google used in their published model trained on the Google news dataset (you can download it from here). The number of features is a "hyper parameter" that you would just have to tune to your application (that is, try different values and see what yields the best results).

If you look at the rows of this weight matrix, these are actually what will be our word vectors!

So the end goal of all of this is really just to learn this hidden layer weight matrix – the output layer we’ll just toss when we’re done!

Let’s get back, though, to working through the definition of this model that we’re going to train.

Now, you might be asking yourself—“That one-hot vector is almost all zeros… what’s the effect of that?” If you multiply a 1 x 10,000 one-hot vector by a 10,000 x 300 matrix, it will effectively just select the matrix row corresponding to the “1”. Here’s a small example to give you a visual.
This means that the hidden layer of this model is really just operating as a lookup table. The output of the hidden layer is just the “word vector” for the input word.

The Output Layer

The $1 \times 300$ word vector for “ants” then gets fed to the output layer. The output layer is a softmax regression classifier. There’s an in-depth tutorial on Softmax Regression [here](http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip...), but the gist of it is that each output neuron (one per word in our vocabulary!) will produce an output between 0 and 1, and the sum of all these output values will add up to 1.

Specifically, each output neuron has a weight vector which it multiplies against the word vector from the hidden layer, then it applies the function $\exp(x)$ to the result. Finally, in order to get the outputs to sum up to 1, we divide this result by the sum of the results from all 10,000 output nodes.

Here’s an illustration of calculating the output of the output neuron for the word “car”.

Note that neural network does not know anything about the offset of the output word relative to the input word. It does not learn a different set of probabilities for the word before the input versus the word after. To understand the implication, let's say that in our training corpus, every single occurrence of the word 'York' is preceded by the word 'New'. That is, at least according to the training data, there is a 100% probability that 'New' will be in the vicinity of 'York'. However, if we take the 10 words in the vicinity of 'York' and randomly pick one of them, the probability of it being 'New' is not 100%; you may have picked one of the other words in the vicinity.
Intuition

Ok, are you ready for an exciting bit of insight into this network?

If two different words have very similar “contexts” (that is, what words are likely to appear around them), then our model needs to output very similar results for these two words. And one way for the network to output similar context predictions for these two words is if the word vectors are similar. So, if two words have similar contexts, then our network is motivated to learn similar word vectors for these two words! Ta da!

And what does it mean for two words to have similar contexts? I think you could expect that synonyms like “intelligent” and “smart” would have very similar contexts. Or that words that are related, like “engine” and “transmission”, would probably have similar contexts as well.

This can also handle stemming for you – the network will likely learn similar word vectors for the words “ant” and “ants” because these should have similar contexts.

Next Up

You may have noticed that the skip-gram neural network contains a huge number of weights... For our example with 300 features and a vocab of 10,000 words, that’s 3M weights in the hidden layer and output layer each! Training this on a large dataset would be prohibitive, so the word2vec authors introduced a number of tweaks to make training feasible. These are covered in part 2 of this tutorial.

Did you know that the word2vec model can also be applied to non-text data for recommender systems and ad targeting? Instead of learning vectors from a sequence of words, you can learn vectors from a sequence of user actions. Read more about this in my new post here.

Other Resources

I’ve also created a post with links to and descriptions of other word2vec tutorials, papers, and implementations.

Cite

Word2Vec Tutorial Part 2 - Negative Sampling
11 Jan 2017

In part 2 of the word2vec tutorial (here’s part 1), I’ll cover a few additional modifications to the basic skip-gram model which are important for actually making it feasible to train.

When you read the tutorial on the skip-gram model for Word2Vec, you may have noticed something—it’s a huge neural network!

In the example I gave, we had word vectors with 300 components, and a vocabulary of 10,000 words. Recall that the neural network had two weight matrices—a hidden layer and output layer. Both of these layers would have a weight matrix with 300 x 10,000 = 3 million weights each!

Running gradient descent on a neural network that large is going to be slow. And to make matters worse, you need a huge amount of training data in order to tune that many weights and avoid over-fitting. Millions of weights times billions of training samples means that training this model is going to be a beast.

The authors of Word2Vec addressed these issues in their second paper.

There are three innovations in this second paper:

1. Treating common word pairs or phrases as single “words” in their model.
2. Subsampling frequent words to decrease the number of training examples.
3. Modifying the optimization objective with a technique they called “Negative Sampling”, which causes each training sample to update only a small percentage of the model’s weights.

It’s worth noting that subsampling frequent words and applying Negative Sampling not only reduced the compute burden of the training process, but also improved the quality of their resulting word vectors as well.

Word Pairs and “Phrases”

The authors pointed out that a word pair like “Boston Globe” (a newspaper) has a much different meaning than the individual words “Boston” and “Globe”. So it makes sense to treat “Boston Globe”, wherever it occurs in the text, as a single word with its own word vector representation.

You can see the results in their published model, which was trained on 100 billion words from a Google News dataset. The addition of phrases to the model swelled the vocabulary size to 3 million words!

If you’re interested in their resulting vocabulary, I poked around it a bit and published a post on it here. You can also just browse their vocabulary here.

Phrase detection is covered in the “Learning Phrases” section of their paper. They shared their implementation in word2phrase.c—I’ve shared a commented (but otherwise unaltered) copy of this code here.

I don’t think their phrase detection approach is a key contribution of their paper, but I’ll share a little about
it anyway since it’s pretty straightforward.

Each pass of their tool only looks at combinations of 2 words, but you can run it multiple times to get longer phrases. So, the first pass will pick up the phrase “New_York”, and then running it again will pick up “New_York_City” as a combination of “New_York” and “City”.

The tool counts the number of times each combination of two words appears in the training text, and then these counts are used in an equation to determine which word combinations to turn into phrases. The equation is designed to make phrases out of words which occur together often relative to the number of individual occurrences. It also favors phrases made of infrequent words in order to avoid making phrases out of common words like “and the” or “this is”.

You can see more details about their equation in my code comments [here](#).

One thought I had for an alternate phrase recognition strategy would be to use the titles of all Wikipedia articles as your vocabulary.

### Subsampling Frequent Words

In part 1 of this tutorial, I showed how training samples were created from the source text, but I’ll repeat it here. The below example shows some of the training samples (word pairs) we would take from the sentence “The quick brown fox jumps over the lazy dog.” I’ve used a small window size of 2 just for the example. The word highlighted in blue is the input word.

<table>
<thead>
<tr>
<th>Source Text</th>
<th>Training Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>The quick brown fox jumps over the lazy dog.</td>
<td>(the, quick) (the, brown)</td>
</tr>
<tr>
<td>The quick brown fox jumps over the lazy dog.</td>
<td>(quick, the) (quick, brown) (quick, fox)</td>
</tr>
<tr>
<td>The quick brown fox jumps over the lazy dog.</td>
<td>(brown, the) (brown, quick) (brown, fox) (brown, jumps)</td>
</tr>
<tr>
<td>The quick brown fox jumps over the lazy dog.</td>
<td>(fox, quick) (fox, brown) (fox, jumps) (fox, over)</td>
</tr>
</tbody>
</table>

There are two “problems” with common words like “the”:

1. When looking at word pairs, (“Fox”, “the”) doesn’t tell us much about the meaning of “fox”. “the” appears in the context of pretty much every word.

2. We will have many more samples of (“the”, …) than we need to learn a good vector for “the”.

http://mccormickml.com/2017/01/11/word2vec-tutorial-part-2-...
Word2Vec implements a “subsampling” scheme to address this. For each word we encounter in our training text, there is a chance that we will effectively delete it from the text. The probability that we cut the word is related to the word’s frequency.

If we have a window size of 10, and we remove a specific instance of “the” from our text:

1. As we train on the remaining words, “the” will not appear in any of their context windows.
2. We’ll have 10 fewer training samples where “the” is the input word.

Note how these two effects help address the two problems stated above.

Sampling rate
The word2vec C code implements an equation for calculating a probability with which to keep a given word in the vocabulary.

\( w_i \) is the word, \( z(w_i) \) is the fraction of the total words in the corpus that are that word. For example, if the word “peanut” occurs 1,000 times in a 1 billion word corpus, then \( z(\text{peanut}) = 1E-6 \).

There is also a parameter in the code named ‘sample’ which controls how much subsampling occurs, and the default value is 0.001. Smaller values of ‘sample’ mean words are less likely to be kept.

\( P(w_i) \) is the probability of keeping the word:

\[
P(w_i) = \left( \frac{z(w_i)}{0.001} + 1 \right) \cdot \frac{0.001}{z(w_i)}
\]

You can plot this quickly in Google to see the shape.

Graph for \((\sqrt{x/0.001} + 1) \cdot 0.001/x\)

![Graph](http://mccormickml.com/2017/01/11/word2vec-tutorial-part-2-...)

No single word should be a very large percentage of the corpus, so we want to look at pretty small values on the x-axis.

Here are some interesting points in this function (again this is using the default sample value of 0.001).

- \( P(w_i) = 1.0 \) (100% chance of being kept) when \( z(w_i) \leq 0.0026 \).
• This means that only words which represent more than 0.26% of the total words will be subsampled.

• \( P(w_i) = 0.5 \) (50% chance of being kept) when \( z(w_i) = 0.00746 \).

• \( P(w_i) = 0.033 \) (3.3% chance of being kept) when \( z(w_i) = 1.0 \).

  ○ That is, if the corpus consisted entirely of word \( w_i \), which of course is ridiculous.

You may notice that the paper defines this function a little differently than what's implemented in the C code, but I figure the C implementation is the more authoritative version.

**Negative Sampling**

Training a neural network means taking a training example and adjusting all of the neuron weights slightly so that it predicts that training sample more accurately. In other words, each training sample will tweak *all* of the weights in the neural network.

As we discussed above, the size of our word vocabulary means that our skip-gram neural network has a tremendous number of weights, all of which would be updated slightly by every one of our billions of training samples!

Negative sampling addresses this by having each training sample only modify a small percentage of the weights, rather than all of them. Here’s how it works.

When training the network on the word pair (“fox”, “quick”), recall that the “label” or “correct output” of the network is a one-hot vector. That is, for the output neuron corresponding to “quick” to output a 1, and for *all* of the other thousands of output neurons to output a 0.

With negative sampling, we are instead going to randomly select just a small number of “negative” words (let’s say 5) to update the weights for. (In this context, a “negative” word is one for which we want the network to output a 0 for). We will also still update the weights for our “positive” word (which is the word “quick” in our current example).

The paper says that selecting 5-20 words works well for smaller datasets, and you can get away with only 2-5 words for large datasets.

Recall that the output layer of our model has a weight matrix that’s 300 x 10,000. So we will just be updating the weights for our positive word (“quick”), plus the weights for 5 other words that we want to output 0. That’s a total of 6 output neurons, and 1,800 weight values total. That’s only 0.06% of the 3M weights in the output layer!

In the hidden layer, only the weights for the input word are updated (this is true whether you’re using Negative Sampling or not).

**Selecting Negative Samples**
The “negative samples” (that is, the 5 output words that we’ll train to output 0) are selected using a “unigram distribution”, where more frequent words are more likely to be selected as negative samples.

For instance, suppose you had your entire training corpus as a list of words, and you chose your 5 negative samples by picking randomly from the list. In this case, the probability for picking the word “couch" would be equal to the number of times “couch” appears in the corpus, divided the total number of word occur in the corpus. This is expressed by the following equation:

\[ P(w_i) = \frac{f(w_i)}{\sum_{j=0}^{n} (f(w_j))} \]

The authors state in their paper that they tried a number of variations on this equation, and the one which performed best was to raise the word counts to the 3/4 power:

\[ P(w_i) = \frac{f(w_i)^{3/4}}{\sum_{j=0}^{n} (f(w_j)^{3/4})} \]

If you play with some sample values, you’ll find that, compared to the simpler equation, this one has the tendency to increase the probability for less frequent words and decrease the probability for more frequent words.

The way this selection is implemented in the C code is interesting. They have a large array with 100M elements (which they refer to as the unigram table). They fill this table with the index of each word in the vocabulary multiple times, and the number of times a word’s index appears in the table is given by \( P(w_i) \) * table_size. Then, to actually select a negative sample, you just generate a random integer between 0 and 100M, and use the word at that index in the table. Since the higher probability words occur more times in the table, you’re more likely to pick those.

**Other Resources**

For the most detailed and accurate explanation of word2vec, you should check out the C code. I’ve published an extensively commented (but otherwise unaltered) version of the code [here](http://mccormickml.com/2017/01/11/word2vec-tutorial-part-2-negative-sampling/).

Also, did you know that the word2vec model can also be applied to non-text data for recommender systems and ad targeting? Instead of learning vectors from a sequence of words, you can learn vectors from a sequence of user actions. Read more about this in my new post [here](http://mccormickml.com/2017/01/11/word2vec-tutorial-part-2-negative-sampling/).

Finally, I’ve also created a [post](http://mccormickml.com/2017/01/11/word2vec-tutorial-part-2-negative-sampling/) with links to and descriptions of other word2vec tutorials, papers, and implementations.

**Cite**