Created by Xodo PDF Studio. Learn more at xodo.com/pdfstudio

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.

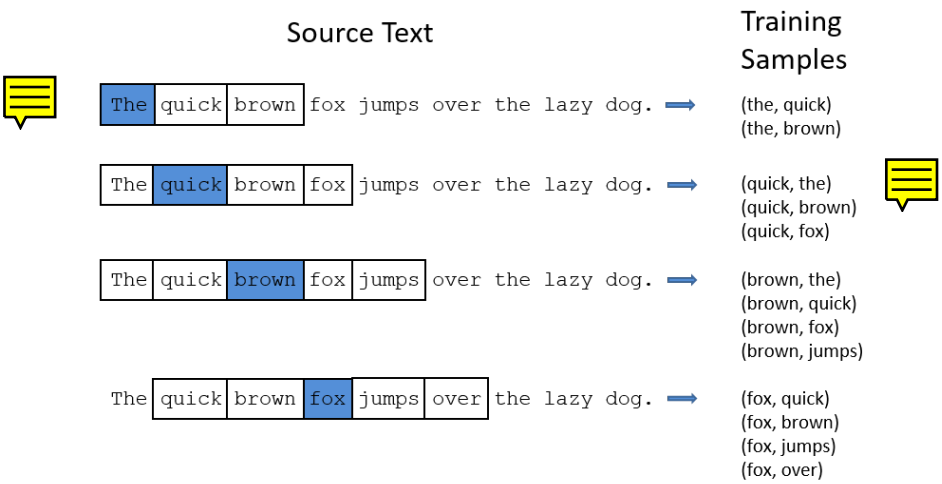
> When I say "nearby", there is actually a "window size" parameter to the algorithm. A typical window size might be 5, meaning 5 words behind and 5 words ahead (10 in total).

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

Created by Xodo PDF Studio. Learn more at xodo.com/pdf-studio

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.



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”.

Want to know
the inner workings of word2vec?
Hey, I’m Chris McCormick. I’m determined to help you master word2vec. My only question is, will you accept my help?
CONTINUE

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.

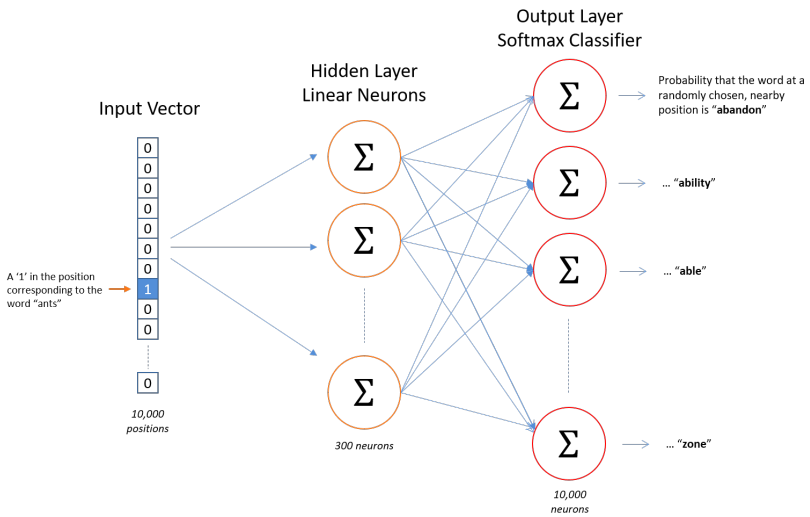
We’re going to represent an input word like “ants” as a one-hot vector. This vector will have 10,000 components (one for every word in our vocabulary)

Created by Xodo PDF Studio. Learn more at xodo.com/pdf-studio

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.

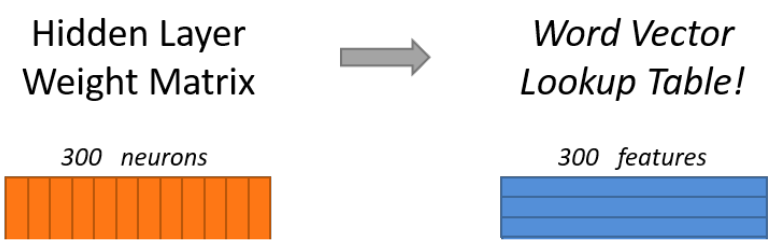
When *training* this network on word pairs, the input is a one-hot vector representing the input word and the training output *is also a one-hot vector* representing the output word. But when you evaluate the trained network on an input word, the output vector will actually be a probability distribution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

The Hidden Layer

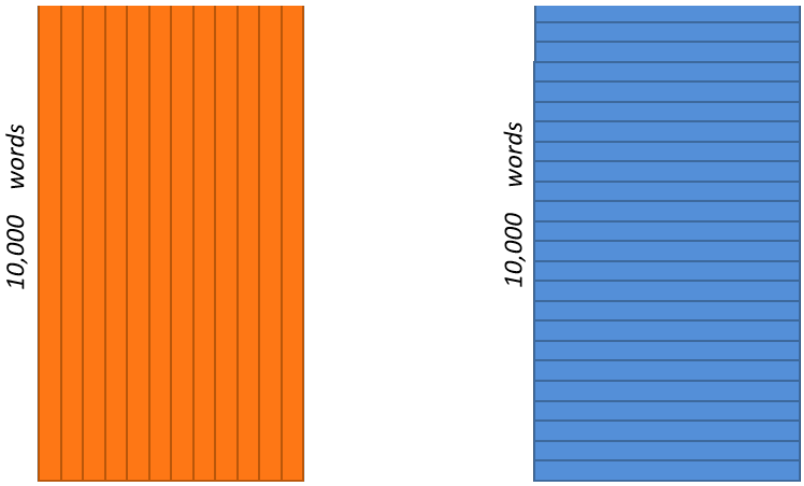
For our example, we're going to say that we're learning word vectors with 300 features. So the hidden layer is going to be represented by a weight matrix with 10,000 rows (one for every word in our vocabulary) and 300 columns (one for every hidden neuron).

> 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!



Created by Xodo PDF Studio. Learn more at xodo.com/pdf-studio



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.

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

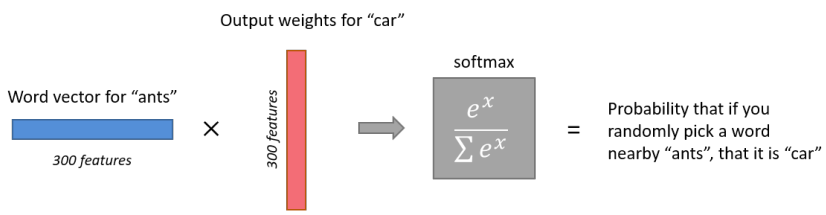
This means that the hidden layer of this model is really just operating as a lookup table. The output of the hidden layer is just the "word vector" for the input word.

The Output Layer

The `1 x 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, 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".



Created by Yodo PDF Studio. Learn more at yodo.com/pdf-studio

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.

Other Resources

Want to know
the inner workings of word2vec?
Hey, I'm Chris McCormick. I'm determined to help you master word2vec. My only question is, will you accept my help?
CONTINUE

Created by Xodo PDF Studio. Learn more at xodo.com/pdfstudio

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.

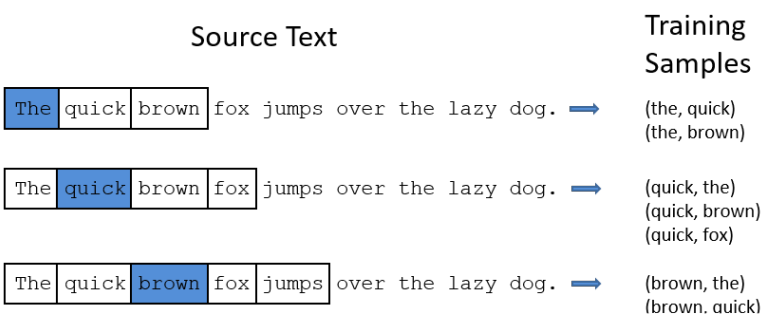
The authors of Word2Vec addressed these issues in their second paper with the following two innovations:

1. Subsampling frequent words to decrease the number of training examples.
2. 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.

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.



Created by Xodo PDF Studio. Learn more at xodo.com/pdf-studio

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".

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.

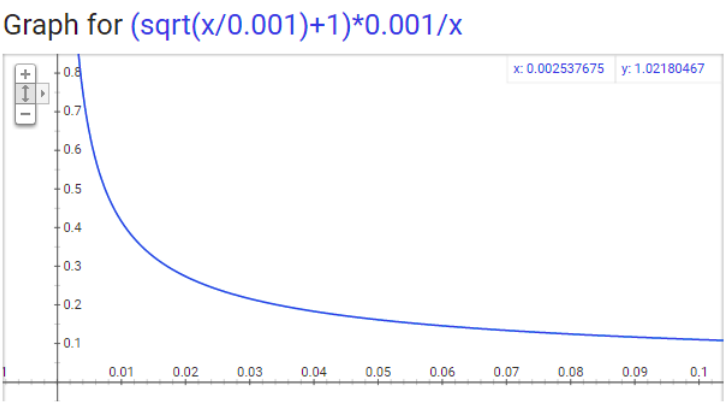
w_i is the word, $z(w_i)$ is the fraction of the total words in the corpus that are that word. For example, if the word "peanut" occurs 1,000 times in a 1 billion word corpus, then z('peanut') = 1E-6.

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:

$$P(w_i) = (\sqrt{\frac{z(w_i)}{0.001}} + 1) \cdot \frac{0.001}{z(w_i)}$$

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



Created by Xodo PDF Studio. Learn more at xodo.com/pdfstudio

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 $z(w_i) <= 0.0026$.
  - 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!

Created by Xodo PDF Studio. Learn more at xodo.com/pdfstudio

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 occus 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:

$$P(w_i) = \frac{f(w_i)^{3/4}}{\sum_{j=0}^{n} \left(f(w_j)^{3/4}\right)}$$

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.

Word Pairs and "Phrases"

The second word2vec paper also includes one more innovation worth discussing. 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.

Created by Yodo PDF Studio. Learn more at yodo.com/pdf-studio

Alex Minnaar

Word2Vec Tutorial Part I: The Skip-Gram Model

12 Apr 2015

In many natural language processing tasks, words are often represented by their *tf-idf* scores. While these scores give us some idea of a word's relative importance in a document, they do not give us any insight into its semantic meaning. *Word2Vec* is the name given to a class of neural network models that, given an unlabelled training corpus, produce a vector for each word in the corpus that encodes its semantic information. These vectors are usefull for two main reasons.

- We can measure the semantic similarity between two words by calculating the cosine similarity between their corresponding word vectors.
- We can use these word vectors as features for various supervised NLP tasks such as document classification, named entity recognition, and sentiment analysis. The semantic information that is contained in these vectors make them powerful features for these tasks.

You may ask *"how do we know that these vectors effectively capture the semantic meanings of the words?"*. The answer is because the vectors adhere surprisingly well to our intuition. For instance, words that we know to be synonyms tend to have similar vectors in terms of cosine similarity and antonyms tend to have dissimilar vectors. Even more surprisingly, word vectors tend to obey the laws of analogy. For example, consider the analogy *"Woman is to queen as man is to king"*. It turns out that

$$v_{queen} - v_{woman} + v_{man} \approx v_{king}$$

where v_{queen},v_{woman},v_{man}, and v_{king} are the word vectors for $queen$, $woman$, man, and $king$ respectively. These observations strongly suggest that word vectors encode valuable semantic information about the words that they represent.

In this series of blog posts I will describe the two main *Word2Vec* models - the *skip-gram model* and the *continuous bag-of-words* model.

Both of these models are simple neural networks with one hidden layer. The word vectors are learned via backpropagation and stochastic gradient descent both of which I descibed in my previous Deep Learning Basics blog post.

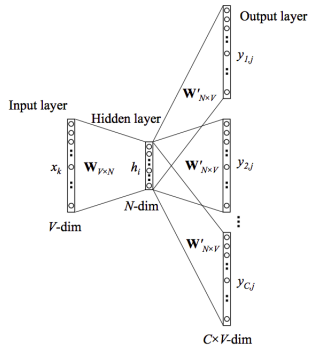
The Skip-Gram Model

Before we define the *skip-gram* model, it would be instructive to understand the format of the training data that it accepts. The input of the *skip-gram* model is a single word w_I and the output is the words in w_I's context $\{w_{O,1},\ldots,w_{O,C}\}$ defined by a word window of size C. For example, consider the sentence *"I drove my car to the store"*. A potential training instance could be the word "car" as an input and the words {"I","drove","my","to","the","store"} as outputs. All of these words are *one-hot* encoded meaning they are vectors of length V (the size of the vocabulary) with a value of 1 at the index corresponding to the word and zeros in all other indexes. As you can see, we are essentially *creating* training examples from plain text which means that we can have a virtually unlimited number of training examples at our disposal.

Forward Propagation

Now let's define the *skip-gram* nerual network model as follows.

Created by Xodo PDF Studio. Learn more at xodo.com/pdf-studio



In the above model $\mathbf{x}$ represents the *one-hot* encoded vector corresponding to the input word in the training instance and $\{\mathbf{y_1}, \ldots \mathbf{y_C}\}$ are the *one-hot* encoded vectors corresponding to the output words in the training instance. The $V \times N$ matrix $\mathbf{W}$ is the weight matrix between the input layer and hidden layer whose i^{th} row represents the weights corresponding to the ith word in the vocabulary. This weight matrix $\mathbf{W}$ is what we are interested in learning because it contains the vector encodings of all of the words in our vocabulary (as its rows). Each output word vector also has an associated $N \times V$ output matrix $\mathbf{W'}$. There is also a hidden layer consisting of N nodes (the exact size of N is a training parameter).

From my previous blog post, we know that the input to a unit in the hidden layer h_i is simply the weighted sum of its inputs. Since the input vector $\mathbf{x}$ is *one-hot* encoded, the weights coming from the nonzero element will be the only ones contributing to the hidden layer. Therefore, for the input $\mathbf{x}$ with $x_k = 1$ and $x_{k'} = 0$ for all $k' \neq k$ the outputs of the hidden layer will be equivalent to the kth row of $\mathbf{W}$. Or mathematically,

$$\mathbf{h} = \mathbf{x}^T\mathbf{W} = \mathbf{W}_{(k,.)} := \mathbf{v}_{w_I}$$

Notice that there is no activation function used here. This is presumably because the inputs are bounded by the *one-hot* encoding. In the same way, the inputs to each of the $C \times V$ output nodes is computed by the weighted sum of its inputs. Therefore, the input to the jth node of the cth output word is

$$u_{c,j} = \mathbf{v}'^T_{w_j}\mathbf{h}$$

However we can also observe that the output layers for each output word share the same weights therefore $u_{c,j} = u_j$. We can finally compute the output of the jth node of the cth output word via the *softmax* function which produces a multinomial distribution

$$p(w_{c,j} = w_{O,c}|w_I) = y_{c,j} = \frac{exp(u_{c,j})}{\sum_{j'=1}^{V} exp(u_{j'})}$$

In plain english, this value is the probability that the output of the jth node of the cth output word is equal to the actual value of the jth index of the cth output vector (which is *one-hot* encoded).

Learning the Weights with Backpropagation and Stochastic Gradient Descent

Now that we know how inputs are propograted forward through the network to produce outputs, we can derive the error gradients necessary for the backpropagation algorithm which we will use to learn both $\mathbf{W}$ and $\mathbf{W'}$. The first step in deriving the gradients is defining a loss function. This loss function will be

$$\begin{aligned} E &= -\log p(w_{O,1}, w_{O,2}, \ldots, w_{O,C}|w_I) \\ &= -\log \prod_{c=1}^{C} \frac{exp(u_{c,j_c^*})}{\sum_{j'=1}^{V} exp(u'_j)} \\ &= -\sum_{c=1}^{C} u_{j_c^*} + C \cdot \log \sum_{j'=1}^{V} exp(u_{j'}) \end{aligned}$$

which is simply the probability of the output words (the words in the input word's context) given the input word. Here, j_c^* is the index of the cth output

Created by Yodo PDF Studio. Learn more at yodo.com/pdf-studio

word. If we take the derivative with respect to $u_{c,j}$ we get

$$\frac{\partial E}{\partial u_{c,j}} = y_{c,j} - t_{c,j}$$

where $t_{c,j} = 1$ if the jth node of the cth true output word is equal to 1 (from its *one-hot* encoding), otherwise $t_{c,j} = 1$. This is the prediction error for node c, j (or the jth node of the cth output word).

Now that we have the error derivative with respect to inputs of the final layer, we can derive the derivative with respect to the output matrix $\mathbf{W'}$. Here we use the chain rule

$$\begin{aligned}\frac{\partial E}{\partial w'_{ij}} &= \sum_{c=1}^{C} \frac{\partial E}{\partial u_{c,j}} \cdot \frac{\partial u_{c,j}}{\partial w'_{ij}} \\ &= \sum_{c=1}^{C} (y_{c,j} - t_{c,j}) \cdot h_i\end{aligned}$$

Therefore the gradient descent update equation for the output matrix $\mathbf{W'}$ is

$$w'^{(new)}_{ij} = w'^{(old)}_{ij} - \eta \cdot \sum_{c=1}^{C} (y_{c,j} - t_{c,j}) \cdot h_i$$

Now we can derive the update equation for the input-hidden layer weights in $\mathbf{W}$. Let's start by computing the error derivative with respect to the hidden layer.

$$\begin{aligned}\frac{\partial E}{\partial h_i} &= \sum_{j=1}^{V} \frac{\partial E}{\partial u_j} \cdot \frac{\partial u_j}{\partial h_i} \\ &= \sum_{j=1}^{V} \sum_{c=1}^{C} (y_{c,j} - t_{c,j}) \cdot w'_{ij}\end{aligned}$$

Now we are able to compute the derivative with respect to $\mathbf{W}$

$$\begin{aligned}\frac{\partial E}{\partial w_{ki}} &= \frac{\partial E}{\partial h_i} \cdot \frac{\partial h_i}{\partial w_{ki}} \\ &= \sum_{j=1}^{V} \sum_{c=1}^{C} (y_{c,j} - t_{c,j}) \cdot w'_{ij} \cdot x_k\end{aligned}$$

and finally we arrive at our gradient descent equation for our input weights

$$w^{(new)}_{ij} = w^{(old)}_{ij} - \eta \cdot \sum_{j=1}^{V} \sum_{c=1}^{C} (y_{c,j} - t_{c,j}) \cdot w'_{ij} \cdot x_j$$

As you can see, each gradient descent update requires a sum over the entire vocabulary V which is computationally expensive. In practice, computation techniques such as hierarchical softmax and negative sampling are used to make this computation more efficient.

References

- Word2Vec Tutorial Part II: The Continuous Bag-of-Words Model
- Distributed Representations of Words and Phrases and their Compositionality, Mikolov et al.
- Natural Language Processing (almost) from Scratch, Collobert et al.

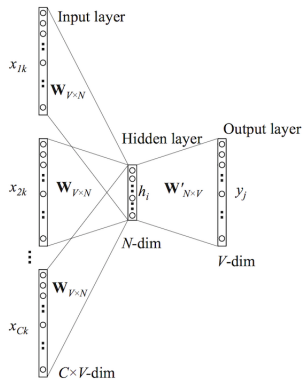
Created by Xodo PDF Studio. Learn more at xodo.com/pdf-studio

Alex Minnaar

Word2Vec Tutorial Part II: The Continuous Bag-of-Words Model

18 May 2015

In the previous post the concept of word vectors was explained as was the derivation of the *skip-gram* model. In this post we will explore the other *Word2Vec* model - the *continuous bag-of-words* (CBOW) model. If you understand the *skip-gram* model then the *CBOW* model should be quite straight-forward because in many ways they are mirror images of each other. For instance, if you look at the model diagram



it looks like the *skip-gram* model with the inputs and outputs reversed. The input layer consists of the *one-hot* encoded input context words $\{\mathbf{x_1},\ldots,\mathbf{x_C}\}$ for a word window of size C and vocabulary of size V. The hidden layer is an N-dimensional vector $\mathbf{h}$. Finally, the output layer is output word $\mathbf{y}$ in the training example which is also *one-hot* encoded. The *one-hot* encoded input vectors are connected to the hidden layer via a $V\times N$ weight matrix $\mathbf{W}$ and the hidden layer is connected to the output layer via a $N\times V$ wieght matrix $\mathbf{W'}$.

Forward Propagation

We must first understand how the output is computed from the input (i.e. forward propagation). The following assumes that we know the input and output weight matrices (I will explain how these are actually learned in the next section). The first step is to evaluate the output of the hidden layer $\mathbf{h}$. This is computed by

$$\mathbf{h}=\frac{1}{C}\mathbf{W}\cdot(\sum_{i=1}^{C}\mathbf{x_i})$$

which is the average of the input vectors weighted by the matrix $\mathbf{W}$. It is worth noting that this hidden layer output computation is one of the only differences between the *continuous bag-of-words* model and the *skip-gram* model (in terms of them being mirror images of course). Next we compute the inputs to each node in the output layer

$$u_j=\mathbf{v'_{w_j}}^T\cdot\mathbf{h}$$

where $\mathbf{v'_{w_j}}$ is the jth column of the output matrix $\mathbf{W'}$. And finally we compute the output of the output layer. The output y_j is obtained by passing the input u_j throught the soft-max function.

$$y_j=p(w_{y_j}|w_1,\ldots,w_C)=\frac{\exp(u_j)}{\sum_{j'=1}^{V}\exp(u'_j)}$$

Now that we know how forward propagation works we can learn the weight matrices $\mathbf{W}$ and $\mathbf{W'}$.

Created by Xodo PDF Studio. Learn more at xodo.com/pdf-studio

Learning the Weight Matrices with Backpropagation

In the process of learning the wieght matrices $\mathbf{W}$ and $\mathbf{W'}$, we begin with randomly initialized values. We then sequentially feed training examples into our model and observe the error which is some function of the difference between the expected output and the actual output. We then compute the gradient of this error with respect to the elements of both weight matrices and correct them in the direction of this gradient. This general optimization procedure is known as stochastic gradient descent (or sgd) but the method by which the gradients are derived is known as backpropagation.

The first step is to define the loss function. The objective is to maximize the conditional probability of the output word given the input context, therefore our loss function will be

$$\begin{aligned} E &= -\log p(w_O|w_I) \\ &= -u_{j*} - \log \sum_{j'=1}^{V} \exp(u_{j'}) \\ &= -\mathbf{v_{w_O}}^T \cdot \mathbf{h} - \log \sum_{j'=1}^{V} \exp(\mathbf{v_{w_{j'}}}^T \cdot \mathbf{h}) \end{aligned}$$

Where j^* is the index of the the actual output word. The next step is to derive the update equation for the hidden-output layer weights $\mathbf{W'}$, then derive the weights for the input-hidden layer weights $\mathbf{W}$

Updating the hidden-output layer weights

The first step is to compute the derivative of the loss function E with respect to the input to the jth node in the output layer u_j.

$$\frac{\partial E}{\partial u_j} = y_j - t_j$$

where $t_j = 1$ if $j = j^*$ otherwise $t_j = 0$. This is simply the prediction error of node j in the output layer. Next we take the derivative of E with respect to the output weight w'_{ij} using the chain rule.

$$\begin{aligned} \frac{\partial E}{\partial w'_{ij}} &= \frac{\partial E}{\partial u_j} \cdot \frac{\partial u_j}{\partial w'_{ij}} \\ &= (y_j - t_j) \cdot h_i \end{aligned}$$

Now that we have the gradient with respect to an arbitrary output weight w'_{ij}, we can define the stochastic gradient descent equation.

$$w_{ij}^{'(new)} = w_{ij}^{'(old)} - \eta \cdot (y_j - t_j) \cdot h_i$$

or $\mathbf{v'_{w_j}}^{(new)} = \mathbf{v'_{w_j}}^{(old)} - \eta \cdot (y_j - t_j) \cdot \mathbf{h}$

where $\eta > 0$ is the learning rate.

Updating the input-hidden layer weights

Now let's try to derive a similar update equation for the input weights w_{ij}. The first step is to compute the derivative of E with respect to an arbitrary hidden node h_i (again using the chain rule).

$$\begin{aligned} \frac{\partial E}{\partial h_i} &= \sum_{j=1}^{V} \frac{\partial E}{\partial u_j} \cdot \frac{\partial u_j}{\partial h_i} \\ &= \sum_{j=1}^{V} (y_j - t_j) \cdot w'_{ij} \end{aligned}$$

where the sum is do to the fact that the hidden layer node h_i is connected to each node of the output layer and therefore each prediction error must be incorporated. The next step is to compute the derivative of E with respect to an arbitrary input weight w_{ki}.

$$\begin{aligned} \frac{\partial E}{\partial w_{ki}} &= \frac{\partial E}{\partial h_i} \cdot \frac{\partial h_i}{\partial w_{ki}} \\ &= \sum_{j=1}^{V} (y_j - t_j) \cdot w'_{ij} \cdot \frac{1}{C} \cdot x_k \\ &= \frac{1}{C} (\mathbf{x} \cdot EH) \end{aligned}$$

Created by Xodo PDF Studio. Learn more at xodo.com/pdf-studio

Where EH is an N-dimensional vector of elements $\sum_{j=1}^{V}(y_j - t_j) \cdot w'_{ij}$ from $i = 1, \ldots, N$. However, since the inputs $\mathbf{x}$ are *one-hot* encoded, only one row of the $N \times V$ matrix $\frac{1}{C}(\mathbf{x} \cdot EH)$ will be nonzero. Thus the final stochastic gradient descent equation for the input weights is

$$\mathbf{v'_{w_{I,c}}}^{(new)} = \mathbf{v'_{w_{I,c}}}^{(old)} - \eta \cdot \frac{1}{C} \cdot EH$$

where $w_{I,c}$ is the cth word in the input context.

References

- Word2Vec Tutorial Part I: The Skip-Gram Model
- Distributed Representations of Words and Phrases and their Compositionality, Mikolov et al.
- Natural Language Processing (almost) from Scratch, Collobert et al.

Created by Xodo PDF Studio. Learn more at xodo.com/pdf-studio

arXiv:1402.3722v1 [cs.CL] 15 Feb 2014

`word2vec` Explained: Deriving Mikolov et al.'s Negative-Sampling Word-Embedding Method

Yoav Goldberg and Omer Levy
{yoav.goldberg,omerlevy}@gmail.com

February 14, 2014

The `word2vec` software of Tomas Mikolov and colleagues[1] has gained a lot of traction lately, and provides state-of-the-art word embeddings. The learning models behind the software are described in two research papers [1, 2]. We found the description of the models in these papers to be somewhat cryptic and hard to follow. While the motivations and presentation may be obvious to the neural-networks language-modeling crowd, we had to struggle quite a bit to figure out the rationale behind the equations.

This note is an attempt to explain equation (4) (*negative sampling*) in "Distributed Representations of Words and Phrases and their Compositionality" by Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado and Jeffrey Dean [2].

1 The skip-gram model

The departure point of the paper is the skip-gram model. In this model we are given a corpus of words w and their contexts c. We consider the conditional probabilities $p(c|w)$, and given a corpus $Text$, the goal is to set the parameters θ of $p(c|w;\theta)$ so as to maximize the corpus probability:

$$\arg\max_{\theta} \prod_{w \in Text} \left[\prod_{c \in C(w)} p(c|w;\theta) \right] \tag{1}$$

in this equation, $C(w)$ is the set of contexts of word w. Alternatively:

$$\arg\max_{\theta} \prod_{(w,c) \in D} p(c|w;\theta) \tag{2}$$

here D is the set of all word and context pairs we extract from the text.

[1] `https://code.google.com/p/word2vec/`

Created by Xodo PDF Studio. Learn more at xodo.com/pdf-studio

1.1 Parameterization of the skip-gram model

One approach for parameterizing the skip-gram model follows the neural-network language models literature, and models the conditional probability $p(c|w;\theta)$ using soft-max:

$$p(c|w;\theta) = \frac{e^{v_c \cdot v_w}}{\sum_{c' \in C} e^{v_{c'} \cdot v_w}} \tag{3}$$

where v_c and $v_w \in R^d$ are vector representations for c and w respectively, and C is the set of all available contexts.[2] The parameters θ are v_{c_i}, v_{w_i} for $w \in V$, $c \in C$, $i \in 1, \cdots, d$ (a total of $|C| \times |V| \times d$ parameters). We would like to set the parameters such that the product (2) is maximized.

Now will be a good time to take the log and switch from product to sum:

$$\arg\max_{\theta} \sum_{(w,c)\in D} \log p(c|w) = \sum_{(w,c)\in D} (\log e^{v_c \cdot v_w} - \log \sum_{c'} e^{v_{c'} \cdot v_w}) \tag{4}$$

An assumption underlying the embedding process is the following:

Assumption maximizing objective 4 will result in good embeddings $v_w \quad \forall\, w \in V$, in the sense that similar words will have similar vectors.

It is not clear to us at this point why this assumption holds.

While objective (4) can be computed, it is computationally expensive to do so, because the term $p(c|w;\theta)$ is very expensive to compute due to the summation $\sum_{c' \in C} e^{v_{c'} \cdot v_w}$ over all the contexts c' (there can be hundreds of thousands of them). One way of making the computation more tractable is to replace the softmax with an *hierarchical softmax*. We will not elaborate on this direction.

2 Negative Sampling

Mikolov et al. [2] present the negative-sampling approach as a more efficient way of deriving word embeddings. While negative-sampling is based on the skip-gram model, it is in fact optimizing a different objective. What follows is the derivation of the negative-sampling objective.

Consider a pair (w, c) of word and context. Did this pair come from the training data? Let's denote by $p(D = 1|w, c)$ the probability that (w, c) came from the corpus data. Correspondingly, $p(D = 0|w, c) = 1 - p(D = 1|w, c)$ will be the probability that (w, c) did not come from the corpus data. As before, assume there are parameters θ controlling the distribution: $p(D = 1|w, c; \theta)$.

[2] Throughout this note, we assume that the words and the contexts come from distinct vocabularies, so that, for example, the vector associated with the word *dog* will be different from the vector associated with the context *dog*. This assumption follows the literature, where it is not motivated. One motivation for making this assumption is the following: consider the case where both the word *dog* and the context *dog* share the same vector v. Words hardly appear in the contexts of themselves, and so the model should assign a low probability to $p(dog|dog)$, which entails assigning a low value to $v \cdot v$ which is impossible.

Created by Xodo PDF Studio. Learn more at xodo.com/pdf-studio

Our goal is now to find parameters to maximize the probabilities that all of the observations indeed came from the data:

$$\begin{aligned}
&\arg\max_{\theta} \prod_{(w,c)\in D} p(D=1|w,c;\theta) \\
&= \arg\max_{\theta} \log \prod_{(w,c)\in D} p(D=1|w,c;\theta) \\
&= \arg\max_{\theta} \sum_{(w,c)\in D} \log p(D=1|w,c;\theta)
\end{aligned}$$

The quantity $p(D=1|c,w;\theta)$ can be defined using softmax:

$$p(D=1|w,c;\theta) = \frac{1}{1+e^{-v_c \cdot v_w}}$$

Leading to the objective:

$$\arg\max_{\theta} \sum_{(w,c)\in D} \log \frac{1}{1+e^{-v_c \cdot v_w}}$$

This objective has a trivial solution if we set θ such that $p(D=1|w,c;\theta)=1$ for every pair (w,c). This can be easily achieved by setting θ such that $v_c = v_w$ and $v_c \cdot v_w = K$ for all v_c, v_w, where K is large enough number (practically, we get a probability of 1 as soon as $K \approx 40$).

We need a mechanism that prevents all the vectors from having the same value, by disallowing some (w,c) combinations. One way to do so, is to present the model with some (w,c) pairs for which $p(D=1|w,c;\theta)$ must be low, i.e. pairs which are not in the data. This is achieved by generating the set D' of random (w,c) pairs, assuming they are all incorrect (the name "negative-sampling" stems from the set D' of randomly sampled negative examples). The optimization objective now becomes:

$$\begin{aligned}
&\arg\max_{\theta} \prod_{(w,c)\in D} p(D=1|c,w;\theta) \prod_{(w,c)\in D'} p(D=0|c,w;\theta) \\
&= \arg\max_{\theta} \prod_{(w,c)\in D} p(D=1|c,w;\theta) \prod_{(w,c)\in D'} (1-p(D=1|c,w;\theta)) \\
&= \arg\max_{\theta} \sum_{(w,c)\in D} \log p(D=1|c,w;\theta) + \sum_{(w,c)\in D'} \log(1-p(D=1|w,c;\theta)) \\
&= \arg\max_{\theta} \sum_{(w,c)\in D} \log \frac{1}{1+e^{-v_c \cdot v_w}} + \sum_{(w,c)\in D'} \log(1-\frac{1}{1+e^{-v_c \cdot v_w}}) \\
&= \arg\max_{\theta} \sum_{(w,c)\in D} \log \frac{1}{1+e^{-v_c \cdot v_w}} + \sum_{(w,c)\in D'} \log(\frac{1}{1+e^{v_c \cdot v_w}})
\end{aligned}$$

Created by Xodo PDF Studio. Learn more at xodo.com/pdf-studio

If we let $\sigma(x) = \frac{1}{1+e^{-x}}$ we get:

$$\begin{aligned}
&\arg\max_{\theta} \sum_{(w,c)\in D} \log \frac{1}{1+e^{-v_c \cdot v_w}} + \sum_{(w,c)\in D'} \log(\frac{1}{1+e^{v_c \cdot v_w}}) \\
&= \arg\max_{\theta} \sum_{(w,c)\in D} \log \sigma(v_c \cdot v_w) + \sum_{(w,c)\in D'} \log \sigma(-v_c \cdot v_w)
\end{aligned}$$

which is almost equation (4) in Mikolov et al ([2]).

The difference from Mikolov et al. is that here we present the objective for the entire corpus $D \cup D'$, while they present it for one example $(w, c) \in D$ and k examples $(w, c_j) \in D'$, following a particular way of constructing D'.

Specifically, with negative sampling of k, Mikolov et al.'s constructed D' is k times larger than D, and for each $(w, c) \in D$ we construct k samples $(w, c_1), \ldots, (w, c_k)$, where each c_j is drawn according to its unigram distribution raised to the 3/4 power. This is equivalent to drawing the samples (w, c) in D' from the distribution $(w, c) \sim p_{words}(w)\frac{p_{contexts}(c)^{3/4}}{Z}$, where $p_{words}(w)$ and $p_{contexts}(c)$ are the unigram distributions of words and contexts respectively, and Z is a normalization constant. In the work of Mikolov et al. each context is a word (and all words appear as contexts), and so $p_{context}(x) = p_{words}(x) = \frac{count(x)}{|Text|}$

2.1 Remarks

- Unlike the Skip-gram model described above, the formulation in this section does not model $p(c|w)$ but instead models a quantity related to the joint distribution of w and c.
- If we fix the words representation and learn only the contexts representation, or fix the contexts representation and learn only the word representations, the model reduces to logistic regression, and is convex. However, in this model the words and contexts representations are learned jointly, making the model non-convex.

3 Context definitions

This section lists some peculiarities of the contexts used in the `word2vec` software, as reflected in the code. Generally speaking, for a sentence of n words $w_1, \ldots, w_n$, contexts of a word w_i comes from a window of size k around the word: $C(w) = w_{i-k}, \ldots, w_{i-1}, w_{i+1}, \ldots, w_{i+k}$, where k is a parameter. However, there are two subtleties:

Dynamic window size the window size that is being used is dynamic – the parameter k denotes the *maximal* window size. For each word in the corpus, a window size k' is sampled uniformly from $1, \ldots, k$.

Created by Xodo PDF Studio. Learn more at xodo.com/pdf-studio

Effect of subsampling and rare-word pruning `word2vec` has two additional parameters for discarding some of the input words: words appearing less than `min-count` times are not considered as either words or contexts, an in addition frequent words (as defined by the `sample` parameter) are down-sampled. Importantly, these words are removed from the text *before* generating the contexts. This has the effect of *increasing the effective window size* for certain words. According to Mikolov et al. [2], sub-sampling of frequent words improves the quality of the resulting embedding on some benchmarks. The original motivation for sub-sampling was that frequent words are less informative. Here we see another explanation for its effectiveness: the effective window size grows, including context-words which are both content-full and linearly far away from the focus word, thus making the similarities more topical.

4 Why does this produce good word representations?

Good question. We don't really know.

The distributional hypothesis states that words in similar contexts have similar meanings. The objective above clearly tries to increase the quantity $v_w \cdot v_c$ for good word-context pairs, and decrease it for bad ones. Intuitively, this means that words that share many contexts will be similar to each other (note also that contexts sharing many words will also be similar to each other). This is, however, very hand-wavy.

Can we make this intuition more precise? We'd really like to see something more formal.

References

[1] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *CoRR*, abs/1301.3781, 2013.

[2] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*, pages 3111–3119, 2013.