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".

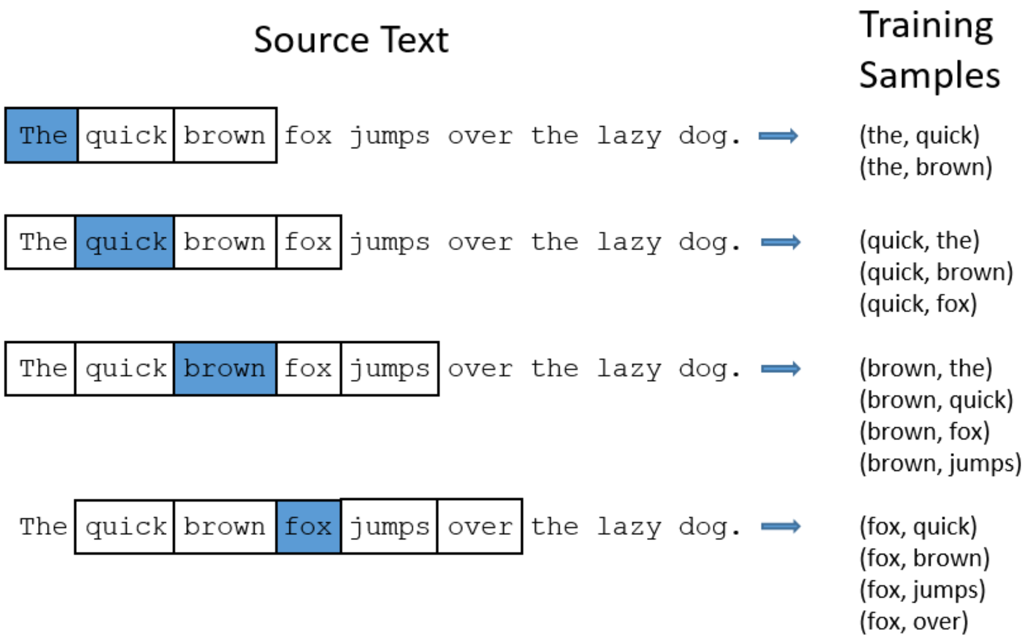
The tool counts the number of times each combination of two words appears in the training text, and then these counts are used in an equation to determine which word combinations to turn into phrases. The equation is designed to make phrases out of words which occur together often relative to the number 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.



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.

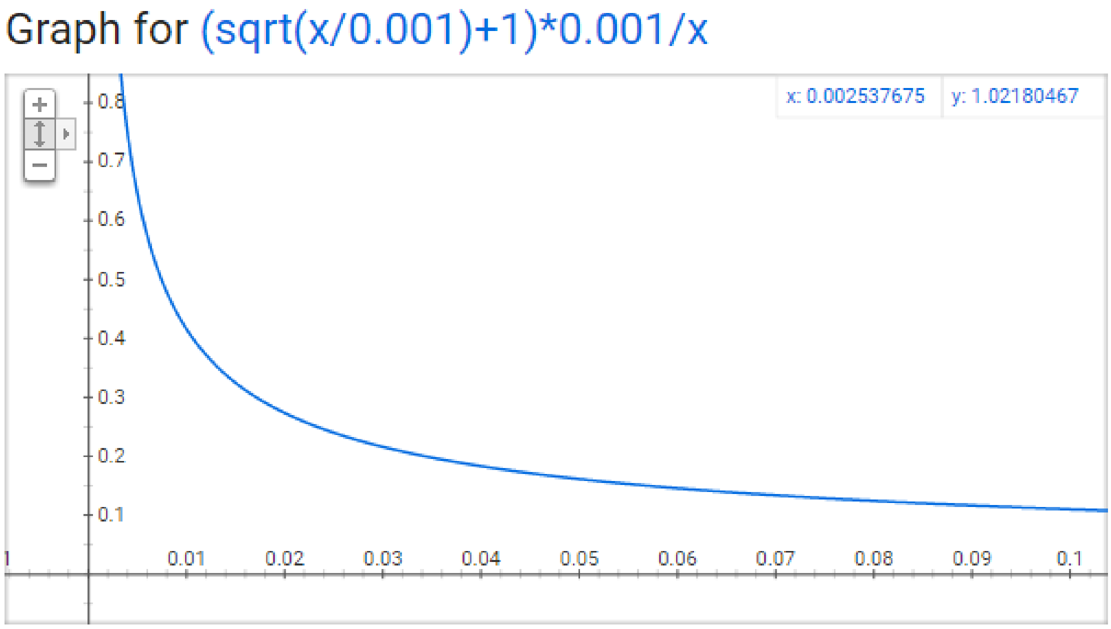
w_i is the word, $z(w_i)$ is the fraction of the total words in the corpus that are that word. For example, if the word “peanut” occurs 1,000 times in a 1 billion word corpus, then z(‘peanut’) = 1E-6.

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.



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!

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 chosen using a "unigram distribution".

Essentially, the probability for selecting a word as a negative sample is related to its frequency, with more frequent words being more likely to be selected as negative samples.

In the word2vec C implementation, you can see the equation for this probability. Each word is given a weight equal to it's frequency (word count) raised to the 3/4 power. The probability for a selecting a word is just it's weight divided by the sum of weights for all words.

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

The decision to raise the frequency to the 3/4 power appears to be empirical; in their paper they say it outperformed other functions. You can look at the shape of the function–just type this into Google: "plot y = x^(3/4) and y = x" and then zoom in on the range x = [0, 1]. It has a slight curve that increases the value a little.

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.

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

Cite

McCormick, C. (2017, January 11). *Word2Vec Tutorial Part 2 - Negative Sampling*. Retrieved from http://www.mccormickml.com

74 Comments **mccormickml.com** 1 **Login**

Recommend 32 Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Yueting Liu • 8 months ago

I have got a virtual map in my head about word2vec within a couple hours thanks to your posts. The concept doesn't seem daunting anymore. Your posts are so enlightening and easily understandable. Thank you so much for the wonderful work!!!

14 ⌃ | ⌄ • Reply • Share ›

Chris McCormick Mod ➦ Yueting Liu • 8 months ago

Awesome! Great to hear that it was so helpful--I enjoy writing these tutorials, and it's very rewarding to hear when they make a difference for people!

3 ⌃ | ⌄ • Reply • Share ›

Jane • 10 months ago

so aweeesome! Thanks Chris! Everything became soo clear! So much fun learn it all!

2 ⌃ | ⌄ • Reply • Share ›

Chris McCormick Mod ➦ Jane • 9 months ago

Haha, thanks, Jane! Great to hear that it was helpful.

⌃ | ⌄ • Reply • Share ›

Ben Bowles • 7 months ago

Thanks for the great tutorial.

About this comment "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!"

Should this actually be 3600 weights total for each training example, given that we have an embedding matrix and an matrix of weights, and BOTH involve updating 1800 weights (300 X 6 neurons)? (Both of which should be whatever dimension you are using for your embeddings multiplied by vocab size)?

1 ⌃ | ⌄ • Reply • Share ›

Chris McCormick Mod ➦ Ben Bowles • 7 months ago

Hi Ben, thanks for the comment.

In my comment I'm talking specifically about the output layer. If you include the hidden layer, then yes, there are more weights updated. The number of weights updated in the hidden layer is only 300, though, not 1800, because there is only a single input word.

So the total for the whole network is 2,100. 300 weights in the hidden layer for the input word, plus 6 x 300 weights in the output layer for the positive word and five negative samples.

layer for the positive word and five negative samples.

And yes, you would replace "300" with whatever dimension you are using for your word embeddings. The vocabulary size does *not* factor into this, though--you're just working with one input word and 6 output words, so the size of your vocabulary doesn't impact this.

Hope that helps! Thanks!

⌃ | ⌄ • Reply • Share ›

Ben Bowles ➦ Chris McCormick • 7 months ago

This is super helpful, I appreciate this. My intuition (however naive it may be) was that the embeddings in the hidden layer for the negative sample words should also be updated as they are relevant to the loss function. Why is this not the case? I suppose I may have to drill down into the equation for backprop to find out. I suppose it has to do with the fact that when the one-hot vector is propagated forward in the network, it amounts to selecting only the embedding that corresponds to the target word.

⌃ | ⌄ • Reply • Share ›

Chris McCormick Mod ➦ Ben Bowles • 7 months ago

That's exactly right--the derivative of the model with respect to the weights of any other word besides our input word is going to be zero.

Hit me up on LinkedIn!

⌃ | ⌄ • Reply • Share ›

Leland Milton Drake ➦ Chris McCormick • 2 months ago

Hey Chris,

When you say that only 300 weights in the hidden layer are updated, are you assuming that the training is done with a minibatch of size 1?

I think if the minibatch is greater than 1, then the number of weights that will be updated in the hidden layer is 300 x number of unique input words in that minibatch.

Please correct me if I am wrong.

And thank you so much for writing this post. It makes reading the academic papers so much easier!

10 ⌃ | ⌄ • Reply • Share ›

Chris McCormick Mod ➦ Leland Milton Drake • a month ago

Hi Leleand, that's correct--I'm just saying that there are only 300 weights updated per input word.

1 ⌃ | ⌄ • Reply • Share ›

Ziyue Jin • 17 days ago

Thank you very much. I am newbie to this area and your visualization helped me a lot. I have a clear picture of why skip-gram model is good.

⌃ | ⌄ • Reply • Share ›

Joey Bose • a month ago

So the subsampling P(w_i) is not really a probability as its not bounded between 0-1. Case in point try it for 1e-6 and you get a 1000 something, threw me for quite a loop when i was coding this.

⌃ | ⌄ • Reply • Share ›

Chris McCormick Mod ➦ Joey Bose • a month ago

Yeah, good point. You can see that in the plot of P(w_i).

⌃ | ⌄ • Reply • Share ›

Robik Shrestha • a month ago

Yet again crystal clear!

⌃ | ⌄ • Reply • Share ›

Chris McCormick Mod ➦ Robik Shrestha • a month ago

Thanks!

⌃ | ⌄ • Reply • Share ›

kasa • 2 months ago

Hi Chris! Great article, really helpful. Keep up the good work. I just wanted to know your opinion on -ve sampling. The reason why we go for backpropagation is to calculate the derivative of Error WRT various weights. If we do -ve sampling, I feel that we are not capturing the true derivative of error entirely; rather we are approximating its value. Is this understanding correct?

⌃ | ⌄ • Reply • Share ›

Vineet John • 3 months ago

Neat blog! Needed a refresher on Negative Sampling and this was perfect.

⌃ | ⌄ • Reply • Share ›

Chris McCormick Mod ➦ Vineet John • 2 months ago

Glad it was helpful, thank you!

1 ⌃ | ⌄ • Reply • Share ›

Jan Chia • 4 months ago

Hi Chris! Thanks for the detailed and clear explanation!

With regards to this portion:
" P(wi)=1.0P(wi)=1.0 (100% chance of being kept) when z(wi)<=0.0026
This means that only words which represent more than 0.26% of the total

This means that only words which represent more than 0.26% of the total words will be subsampled. "

Do you actually mean that only words that represent 0.26% or less will be used?
My understanding of this subsampling is that we want to keep words that appears less frequently.

Do correct me if I'm wrong! :)

Thanks!

^ | ˅ • Reply • Share ›

Chris McCormick Mod ➦ Jan Chia • 3 months ago

You're correct--we want to keep the less frequent words. The quoted section is correct as well, it states that *every instance* of words that represent 0.26% or less will be kept. It's only at higher percentages that we start "subsampling" (discarding some instances of the words).

^ | ˅ • Reply • Share ›

Jan Chia ➦ Chris McCormick • 3 months ago

Ahh! Thank you so much for the clarification!

^ | ˅ • Reply • Share ›

김개미 • 4 months ago

Not knowing negative sampling, InitUnigramTable() makes me confused but i've finally understood the codes from this article. Thank you so much!

^ | ˅ • Reply • Share ›

Malik Rumi • 4 months ago

" I don’t think their phrase detection approach is a key contribution of their paper"
Why the heck not?

^ | ˅ • Reply • Share ›

Ujan Deb • 4 months ago

Thanks for writing such a wonderful article Chris! Small doubt. When you say "1 billion word corpus" in the sub-sampling part, does that mean the number of different words, that is vocabulary size is 1 billion or just the total number of words including repetitions is 1 billion? I'm implementing this from scratch. Thanks.

^ | ˅ • Reply • Share ›

Ujan Deb ➦ Ujan Deb • 4 months ago

Okay after giving it another read I think its the later

^ | ˅ • Reply • Share ›

Derek Osborne • 4 months ago

Just to pile on some more, this is a fantastic explanation of word2vec. I watched the Stanford 224n lecture a few time and could not make sense of what was going on with word2vec. Everything clicked once I read this post. Thank you!

^ | ⌄ • Reply • Share ›

Ujan Deb ➦ Derek Osborne • 4 months ago

Hi Derek. Care to see if you know the answer to my question above ? Thanks.

^ | ⌄ • Reply • Share ›

Leon Ruppen • 4 months ago

wonderfull job, the commented C code is escpecially useful! thanks!

^ | ⌄ • Reply • Share ›

Himanshu Ahuja • 4 months ago

Can you please elaborate this: "In the hidden layer, only the weights for the input word are updated (this is true whether you're using Negative Sampling or not)." Wouldn't the weight of all the samples we randomly selected be tweaked a little bit? Like in the 'No Negative Sampling' case, where all the weights were slightly tweaked a bit.

^ | ⌄ • Reply • Share ›

Anmol Biswas ➦ Himanshu Ahuja • 4 months ago

No actually.
The weight update rule in Matrix terms can be written something like this :
-learning_rate*(Output Error)*(Input vector transposed) [the exact form changes depending on how you define your vectors and matrices]

Looking at the expression, it becomes clear that when your "Input Vector" is a One-hot encoded vector, it will effectively create a Weight Update matrix which has non-zero values only in the respective column (or row, depending on your definitions) where the "Input Vector" has a '1'

^ | ⌄ • Reply • Share ›

Addy R • 4 months ago

Thanks for the post! I have one question, is the sub-sampling procedure to be used along with negative sampling? or does sub-sampling eliminate the need for negative sampling?

^ | ⌄ • Reply • Share ›

Chris McCormick Mod ➦ Addy R • 4 months ago

They are two different techniques with different purposes which unfortunately have very similar names :). Both are implemented and used in Google's implementation--they are not alternatives for each other.

^ | ⌄ • Reply • Share ›

Addy R ➦ Chris McCormick • 4 months ago

Thank you Chris! One other quick query - does the original idea have a special symbol for <start> and <end> of a sentence? I know OOVs are dropped form the data, but what about start and end? This might matter for the cbow model.

^ | ˅ • Reply • Share ›

Manish Chablani • 5 months ago

Such a great explanation. Thank you Chris !!

^ | ˅ • Reply • Share ›

Chris McCormick Mod ➦ Manish Chablani • 5 months ago

Thanks, Manish!

^ | ˅ • Reply • Share ›

Rumesa Farooq • 6 months ago

Hello Chris
I have gone through your commented C code of word2vec at github.It was really nice.But I have a problem.When I run the code with CBOW=1, it creates vectors and then display cousine distance or similarity by entering a word.But when I run it in skipgram mode i.e CBOW=0, it also creates vectors but donot display the cousine distance for any word.Any suggestion? How to use distance file for skipgram?

^ | ˅ • Reply • Share ›

Chris McCormick Mod ➦ Rumesa Farooq • 6 months ago

Sorry, I haven't had a chance to play with the code yet (I just read through it to understand it), so I don't know much about using it.

^ | ˅ • Reply • Share ›

Jay • 6 months ago

Very nice. Would be helpful to add a short section on updating the weights and showing them converge.

^ | ˅ • Reply • Share ›

Chris McCormick Mod ➦ Jay • 6 months ago

Thanks, Jay!

^ | ˅ • Reply • Share ›

Chen Lu • 6 months ago

Thanks Chris, it is really a very useful tutorial. One thing (which may be too naive, I am not able to get yet, is the one-hot vector part, which part of the code provided in the (github/word2vec_commented) is doing that part?

^ | ˅ • Reply • Share ›

Chris McCormick Mod ➦ Chen Lu • 6 months ago

Hi Chen,
The one-hot vector is part of the mathematical formulation of the skip-gram architecture, but in code, you don't need it. All the one-hot vector does is select the corresponding word vector from the middle layer, so you won't find any one-hot vectors in the code.

^ | ⌄ • Reply • Share ›

Mahesh Govind • 7 months ago

Hi Chris,
Thank you for the good tutorial .
A query about visualising word vector. When they visualize the vectors , are we doing a dimensionality reduction and plotting them. And when we plot them , ie the word vectors , those vectors which are similar (dot product value) are displayed around same area.

^ | ⌄ • Reply • Share ›

Chris McCormick Mod ➦ Mahesh Govind • 7 months ago

Hi Mahesh, thanks. I'm not sure how they do the visualizations, but I would guess the same as you, that they're doing something like PCA to reduce to 2 dimensions.

^ | ⌄ • Reply • Share ›

Mahesh Govind ➦ Chris McCormick • 7 months ago

Thank you for the reply . One more foolish query . Word2vec will provide one word co-related to the input word as the result, irrespective of the window size ? The window size will have impact only on creating a number of training tuples ?
RNN is used to provide the context for a sentence . Is this intuition correct

^ | ⌄ • Reply • Share ›

Chris McCormick Mod ➦ Mahesh Govind • 7 months ago

Yes, training is done with one output word at a time. Increasing the window size can affect the resulting word vectors, since it will add farther away words to the context of the input word.

^ | ⌄ • Reply • Share ›

Huichang Han • 8 months ago

Great article, this is the best tutorial for me to learn the insight of word2vec.

^ | ⌄ • Reply • Share ›

Chris McCormick Mod ➦ Huichang Han • 8 months ago

Thank you :)

^ | ⌄ • Reply • Share ›

Walker • 8 months ago

Is tf.nn.sampled_softmax_loss() an implementation of negative sampling?

is tf.nn.sampled_softmax_loss() an implementation of negative sampling?

I always think they are different. Sampled softmax loss is a fast way to compute softmax (because the denominator of the softmax is huge) and negative sampling is ... emm ... conceptually "randomly select just a small number of “negative” words (let’s say 5) to update the weights for", i don't really know how to implement it. It's like a big black box to me. And just now some friend mentioned that we can use sampled softmax loss to implement negative sampling. I was shocked and did a little research about it.

It seems like they are from different paper and https://www.tensorflow.org/... describes that NCE is a generalized version

Related posts

Product Quantizers for k-NN Tutorial Part 2 22 Oct 2017
Product Quantizers for k-NN Tutorial Part 1 13 Oct 2017
k-NN Benchmarks Part I - Wikipedia 08 Sep 2017