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 the all 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.

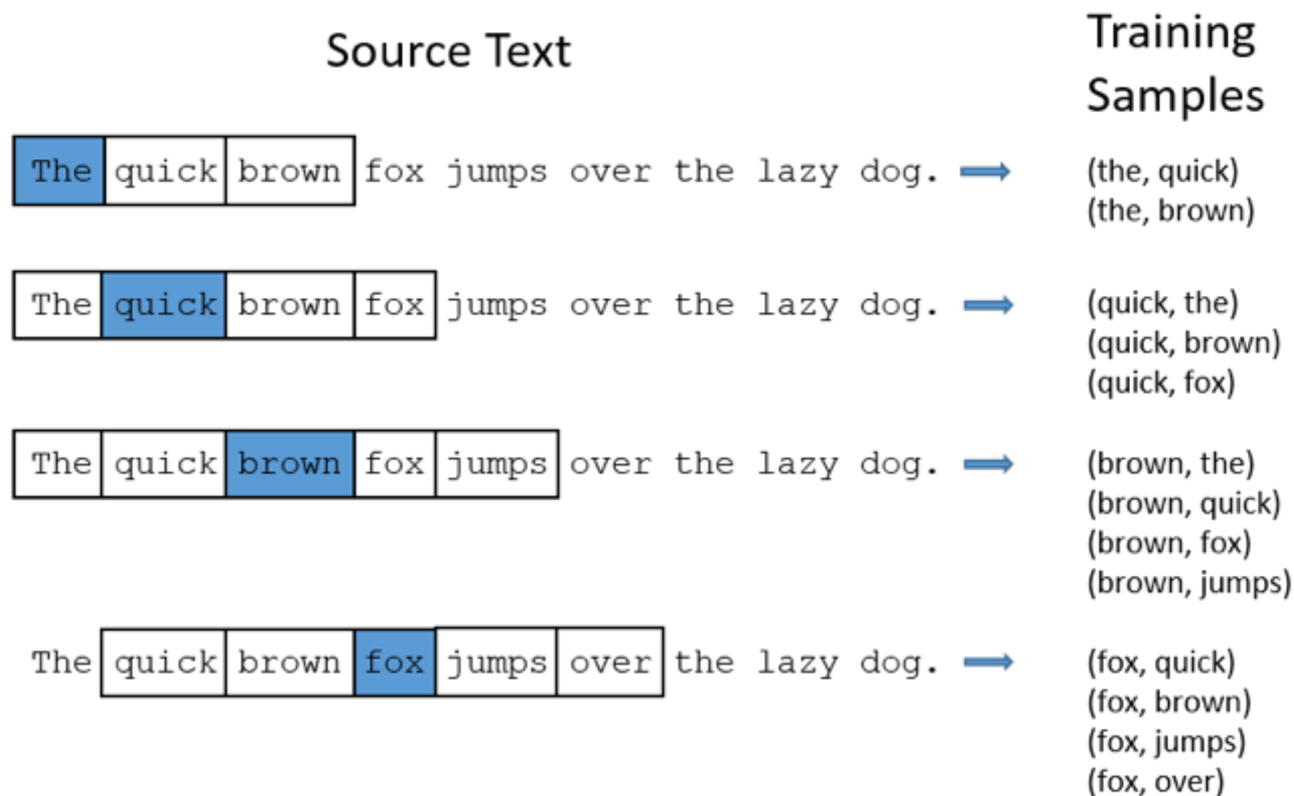
We’re going to train the neural network to do the following. Given a specific word in the middle of a sentence (the input word), look at the words nearby and pick one at random. The network is going to tell us the probability for every word in our vocabulary of being the “nearby word” that we chose.

> 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.



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?

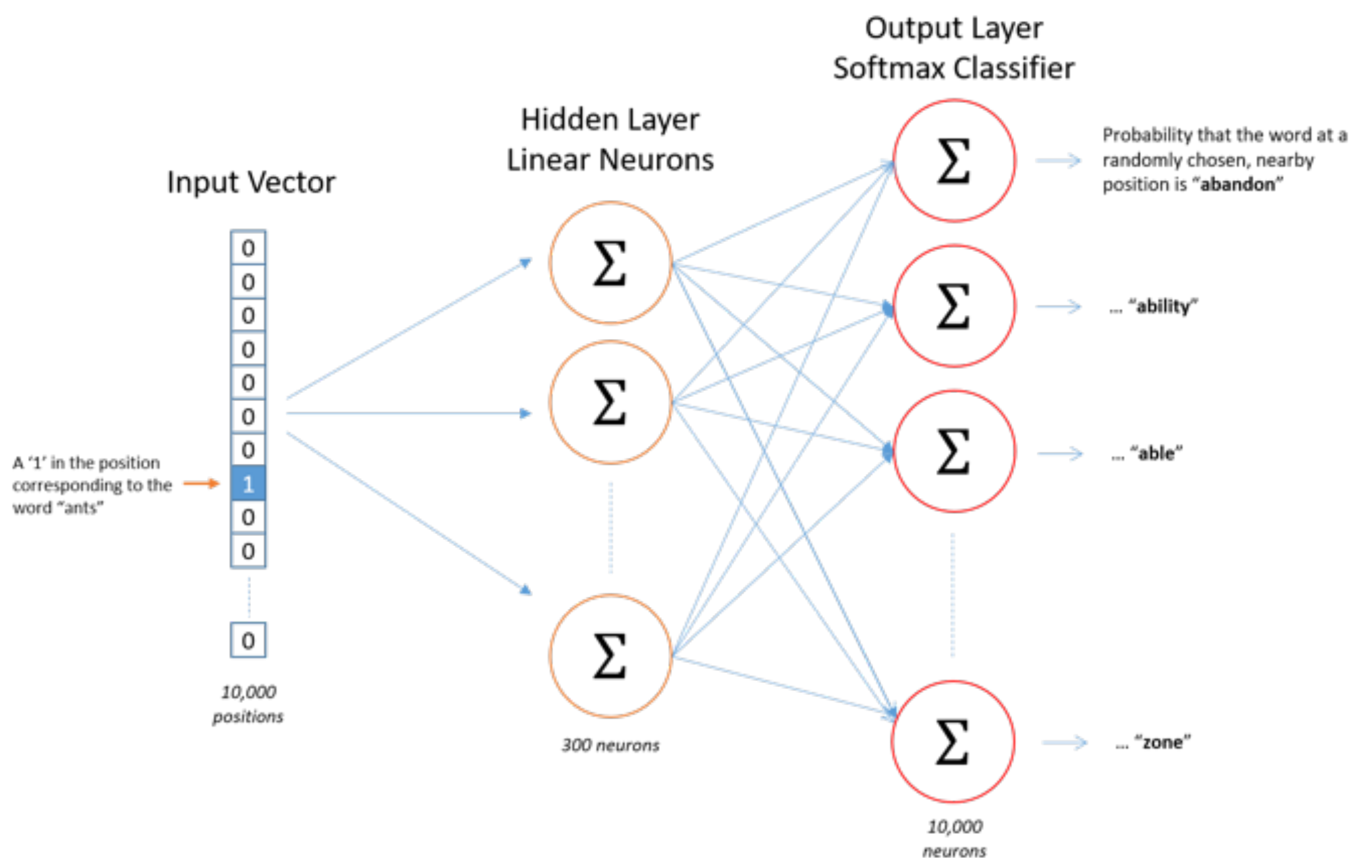
First of all, you know you can't feed a word just as a text string to a neural network, so we need a way to represent the words to the network. To do

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

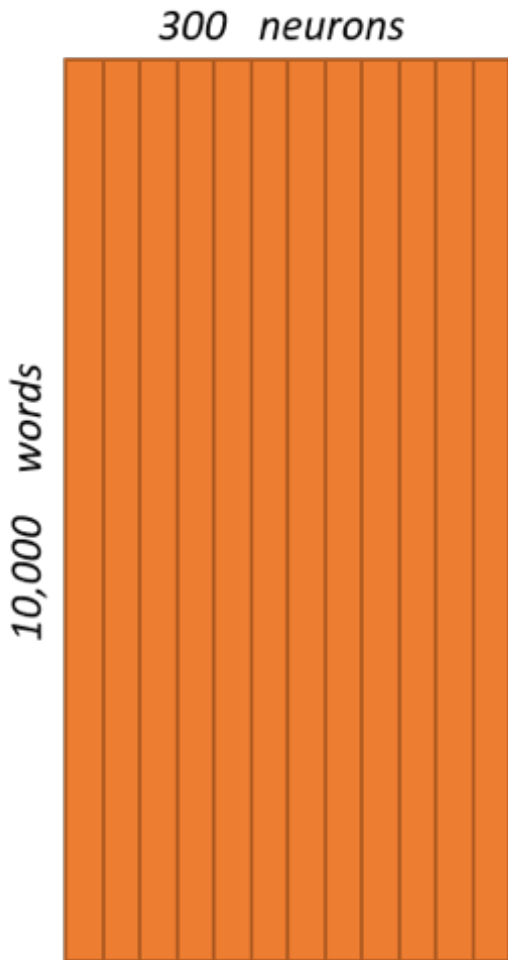
For our example, we're going to say that we're learning word vectors with 300 features. So the hidden layer is going to be represented by a weight matrix with 10,000 rows (one for every word in our vocabulary) and 300 columns (one for every hidden neuron).

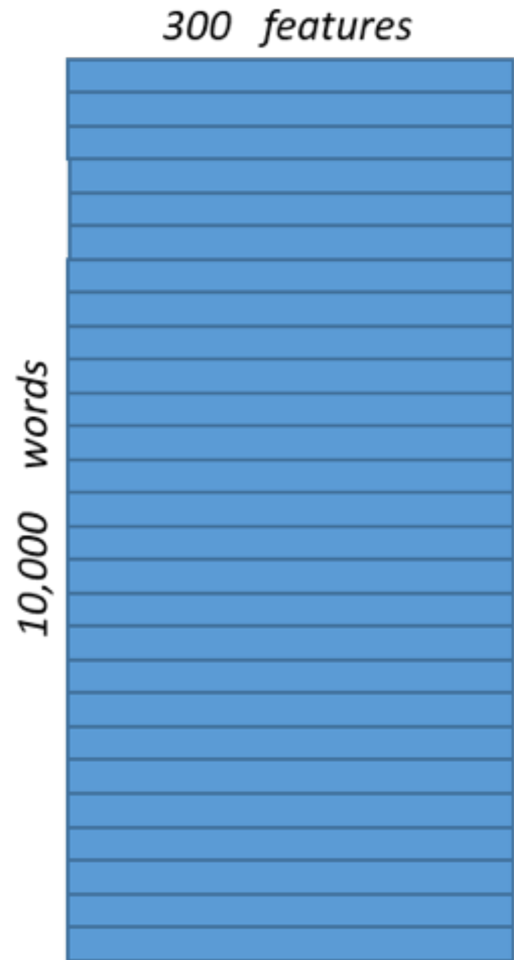
> 300 features is what Google used in their published model trained on the Google news dataset (you can download it from here). The number of features is a "hyper parameter" that you would just have to tune to your application (that is, try different values and see what yields the best results).

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

Hidden Layer Weight Matrix

Word Vector Lookup Table!





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.

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

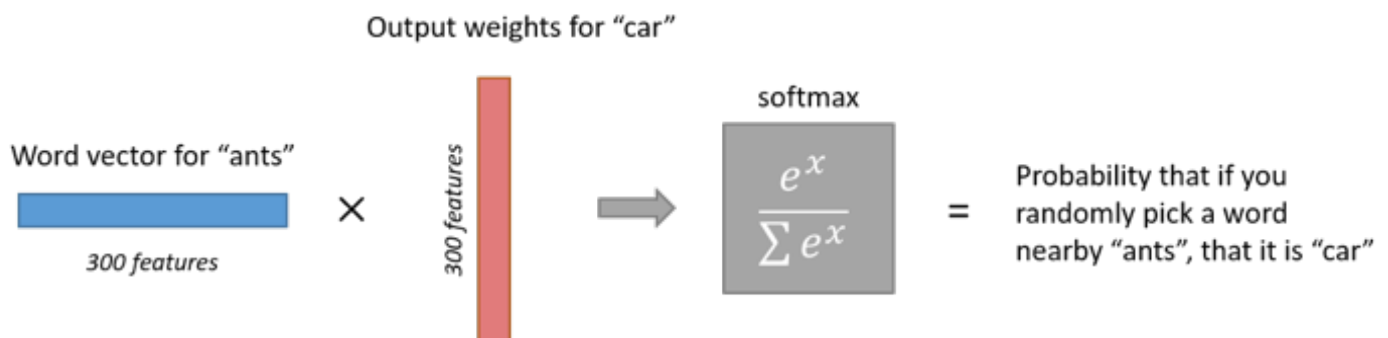
This means that the hidden layer of this model is really just operating as a lookup table. The output of the hidden layer is just the "word vector" for the input word.

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".



> 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

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

Cite

McCormick, C. (2016, April 19). *Word2Vec Tutorial - The Skip-Gram Model*. Retrieved from http://www.mccormickml.com

Deep Learning Computer Vision - 3-Day Class
A hands-on deep-dive into TensorFlow for computer vision, taught by experts! t

204 Comments mccormickml.com

106

Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

zul waker • a year ago

You are amazing. I tried to understand this from so many sources but you gave the best explanation possible. many thanks.

25 • Reply • Share ›

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.

Source Text	Training Samples
The quick brown fox jumps over the lazy dog.	(the, quick) (the, brown)
The quick brown fox jumps over the lazy dog.	(quick, the) (quick, brown) (quick, fox)
The quick brown fox jumps over the lazy dog.	(brown, the) (brown, quick) (brown, fox) (brown, jumps)
The quick brown fox jumps over the lazy dog.	(fox, quick) (fox, brown) (fox, jumps) (fox, over)

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.

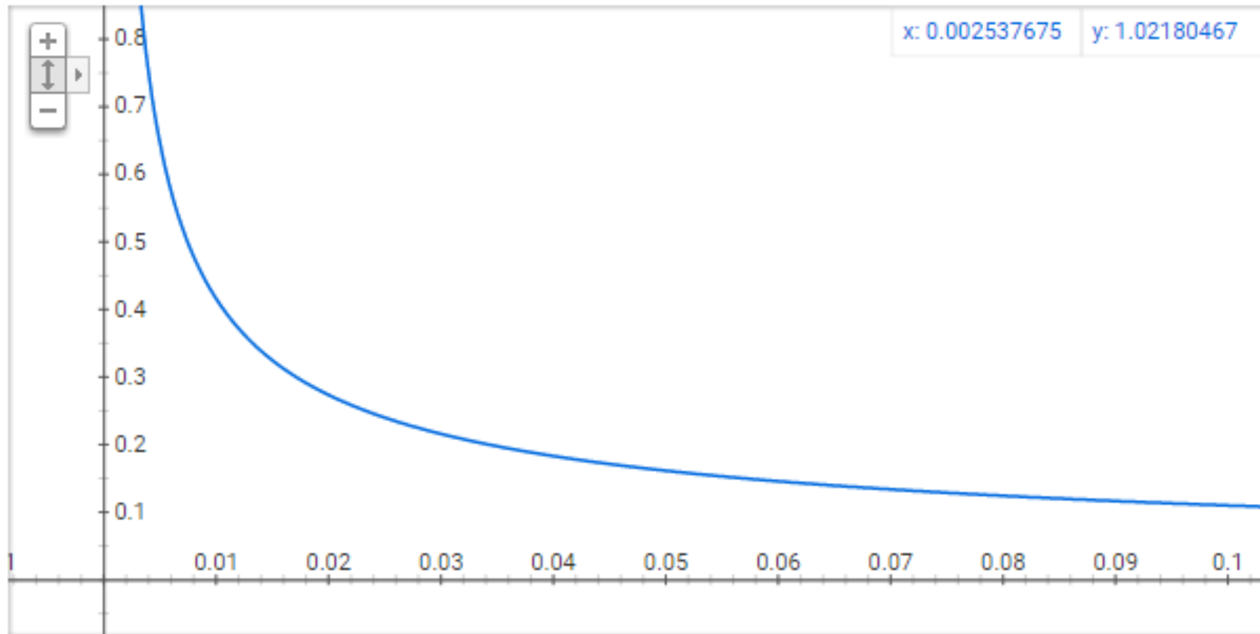
w_i is the word, $z(w_i)$ is the fraction of the total words in the corpus that are that word. For example, if the word “peanut” occurs 1,000 times in a 1 billion word corpus, then z(‘peanut’) = 1E-6.

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

90 Comments mccormickml.com

Recommend 56 Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Yueting Liu • a year 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!!!

22 • Reply • Share ›

Chris McCormick Mod → Yueting Liu • a year 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!

4 • Reply • Share ›

1mike12 → Yueting Liu • 6 months ago

I agree, shit is lit up cuz

1 • Reply • Share ›

Laurence Obi • 4 months ago

Hi Chris,

Awesome post. Very insightful. However, I do have a question. I noticed that in the bid to reduce the amount of weights we'll have to update, frequently occurring words were paired and viewed as one word. Intuitively, that looks to me like we've just added an extra word (New York) while other versions of the word New that have not occurred with the word York would be treated as stand-alone. Am I entirely wrong to assume that the ultimate size of our one-hot encoded vector would grow in this regard? Thanks.

2 • Reply • Share ›

Jane • 2 years ago

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

2 • Reply • Share ›

Chris McCormick Mod → Jane • 2 years ago

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

^ | ⌄ • Reply • Share ›

George Ho • 3 months ago

Incredible post Chris! Really like the way you structured both tutorials: it's so helpful to understand the crux of an algorithm before moving on to the bells and whistles that make it work in practice. Wish more tutorials were like this!

1 ^ | ⌄ • Reply • Share ›

fangchao liu • 4 months ago

Thanks a lot for your awesome blog!
But I got a question about the negative sampling process while reading.
In the paper, it'll sample some negative words to which the outputs are expected zeros, but what if the sampled word is occasionally in the comtext of the input word? Form example, sentence is "The quick brown fox jumps over the lazy dog", input word is "fox", the positive word is "jumps", and one sampled word is "brown". Will this situation result in some errors?

1 ^ | ⌄ • Reply • Share ›

Ben Bowles • a year 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 • a year 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.

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!

^ | v • Reply • Share ›

Ben Bowles → Chris McCormick • a year 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.

^ | v • Reply • Share ›

Chris McCormick Mod → Ben Bowles • a year 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!

^ | v • Reply • Share ›

Leland Milton Drake → Chris McCormick • 10 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!

23 ^ | v • Reply • Share ›

Chris McCormick Mod ➦ Leland Milton Drake • 10 months ago

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

1 ^ | ⌄ • Reply • Share ›

Gabriel Falcones • 20 days ago

Truly a great explanation!! Thanks for the tutorial!!

^ | ⌄ • Reply • Share ›

Rajani M • 2 months ago

Hi. I have doubt on skipgram.

Consider the window size 5 and sentence contains less than 5 words. Whether this kind of sentence trains? or ignores?

^ | ⌄ • Reply • Share ›

Andres Suarez ➦ Rajani M • 2 months ago

It will train with the available words only.

^ | ⌄ • Reply • Share ›

Rajani M ➦ Andres Suarez • 2 months ago

you mean, It will train the words even though sentence contains less than five words, right?

^ | ⌄ • Reply • Share ›

Andres Suarez ➦ Rajani M • 2 months ago

Indeed, it will train a sentence with less words than needed by the window, and it will use only the available words.

^ | ⌄ • Reply • Share ›

Rajani M ➦ Andres Suarez • 2 months ago

Thank you so much. This helped me lot.

^ | ⌄ • Reply • Share ›

adante • 5 months ago

Thank you - this is around the fifth word2vec tutorial I've read and the first that has made sense!

^ | ⌄ • Reply • Share ›

li xiang • 5 months ago

awesome！ i am a student from china ,I had read a lot of papers about word2vec.I still can not understand it. after read your post, i finally figured out the insight of word2vec。thanks a lot。

^ | v • Reply • Share ›

Aakash Tripathi • 6 months ago

A great blog!!
Got a query in the end- In negative sampling, if weights for "more likely" words are tuned more often in output layer, then if a word is very infrequent (ex - it comes only once in the entire corpus), how'd it's weights be updated?

^ | v • Reply • Share ›

Andres Suarez ➦ Aakash Tripathi • 2 months ago

The weights for a word, the word vector itself, is just updated when the word occurs as input. The negative sampling affects the weights of the output layer, which are not considered in the final vector model.
Looking at the implementation, there's also a parameter which cuts down words that occur less times than a threshold (which is a parameter), so a hapax legomenon (a word occuring only once in a corpus) it might not even appear in any training pair, unless you set your threshold to 1.

^ | v • Reply • Share ›

Deepak Sharma • 6 months ago

Amazingly explained! I am grateful.

^ | v • Reply • Share ›

Janina Nuber • 9 months ago

Really nicely explained. Yet I have one question - what is the intuition behind this: "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."

I would have expected quite the opposite - that one needs more negative samples in large datasets than in small datasets. Is it because smaller datasets are more prone to overfitting, so I need more negative samples to compensate for that?

Thanks so much!

^ | v • Reply • Share ›

Ziyue Jin • 9 months 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.

^ | v • Reply • Share ›

Joey Bose • 10 months 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.

^ | v • Reply • Share ›

Chris McCormick Mod ➦ Joey Bose • 10 months ago

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

^ | v • Reply • Share ›

Robik Shrestha • 10 months ago

Yet again crystal clear!

^ | v • Reply • Share ›

Chris McCormick Mod ➦ Robik Shrestha • 10 months ago

Thanks!

^ | v • Reply • Share ›

kasa • a year 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?

^ | v • Reply • Share ›

Vineet John • a year ago

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

^ | v • Reply • Share ›

Chris McCormick Mod ➦ Vineet John • a year ago

Glad it was helpful, thank you!

1 ^ | v • Reply • Share ›

Jan Chia • a year 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 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!

^ | v • Reply • Share ›

Chris McCormick Mod ➦ Jan Chia • a year 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).

^ | v • Reply • Share ›

Jan Chia ➦ Chris McCormick • a year ago

Ahh! Thank you so much for the clarification!

^ | v • Reply • Share ›

김개미 • a year ago

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

^ | v • Reply • Share ›

Malik Rumi • a year ago

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

^ | v • Reply • Share ›

Ujan Deb • a year 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.

^ | v • Reply • Share ›

Ujan Deb ➦ Ujan Deb • a year ago

Okay after giving it another read I think its the later

^ | ⌄ • Reply • Share ›

Derek Osborne • a year 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 • a year ago

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

^ | ⌄ • Reply • Share ›

Leon Ruppen • a year ago

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

^ | ⌄ • Reply • Share ›

Himanshu Ahuja • a year 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 • a year 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'

1 ^ | ⌄ • Reply • Share ›

Addy R • a year 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 • a year 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 • a year 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 • a year ago

Such a great explanation. Thank you Chris !!

^ | ˅ • Reply • Share ›

Load more comments

ALSO ON **MCCORMICKML.COM**

Image Derivative

14 comments • 2 years ago

Aya al-bitar — I've read your blogs about HOG , Gradient vector and image derivative and they were extremely helpful, Thanks ..

Deep Learning Tutorial - Sparse Autoencoder

7 comments • 2 years ago

Choung young jae — in sparse autoencoder paramether rho may be 0.05 not the 0.5Thanks! Tutorial

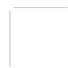
Kernel Regression

2 comments • 2 years ago

LIU VIVIAN — Only this introduction did I get understand this kernel regression. I've got frustration for a period. Thanks so much!

Interpreting LSI Document Similarity

13 comments • 2 years ago



Chris McCormick — Yeah, regular expressions sounds like the right answer!Google has a good introduction to …

✉ Subscribe   D Add Disqus to your siteAdd DisqusAdd   🔒 Disqus' Privacy PolicyPrivacy PolicyPrivacy

Related posts

Applying word2vec to Recommenders and Advertising 15 Jun 2018

Product Quantizers for k-NN Tutorial Part 2 22 Oct 2017

Product Quantizers for k-NN Tutorial Part 1 13 Oct 2017

© 2018. All rights reserved.

Chris McCormick Mod ↪ zul waker • a year ago

Thanks so much, really glad it helped!

4 ^ | v • Reply • Share ›

raj1514 • a year ago

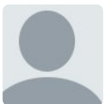
Thanks for this post! It really saved time in going through papers about this...

23 ^ | v • Reply • Share ›

Chris McCormick Mod ↪ raj1514 • a year ago

Great! Glad it helped.

^ | v • Reply • Share ›



ningyuwhut • 6 months ago

I have a question that how to understand skip in the name "the Skip-Gram Model" literally? I mean why this model called the skip-gram model. Thanks

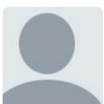
17 ^ | v • Reply • Share ›

Supun Abeysinghe ↪ ningyuwhut • 6 months ago

Before this there was a bi-gram model which uses the most adjacent word to train the model. But in this case the word can be any word inside the window. So you can use any of the words inside the window skipping the most adjacent word. Hence skip-gram.

I'm not sure though :)

1 ^ | v • Reply • Share ›



micsca • 2 years ago

nice article!

9 ^ | v • Reply • Share ›

Arish Ali • a year ago

Loved the simplicity of the article and the visualizations of different numeric operations made it so easy to understand

6 ^ | v • Reply • Share ›

Chris McCormick Mod ↪ Arish Ali • a year ago

Awesome, thanks!

^ | v • Reply • Share ›

Albert Wang • a year ago

The best word2vec tutorial I have ever read besides the paper.

One question:
Since the algorithm knows nothing about the slicing window, does that mean there is no difference between the first word after the target word and the second word after the target word?

For example, if the window is [I am a software engineer], here the target word is "a".

The algorithm will train the neural network 4 times. Each time, the output will be a softmax vector and it computes the cross entropy loss between the output vector and the true one-hot vector which represents "i", "am", "software", and "engineer".

Therefore, this is just a normal softmax classifier. But word2vec uses it in a smart way.

Do they use "cross entropy"? Which loss function do they use?

4 ^ | v • Reply • Share ›

Chris McCormick Mod → Albert Wang • a year ago

Hi Albert,

You're correct that the position of the word within the context window has no impact on the training.

I'm hesitant to answer you question about the cost function because I'm not familiar with variations of the softmax classifier, but I believe you're correct that it's an ordinary softmax classifier like [here](http://ufldl.stanford.edu/t....

To reduce the compute load they do modify the cost function a bit with something called Negative Sampling--read about that in part 2 of this tutorial.

^ | v • Reply • Share ›

Albert Wang → Chris McCormick • a year ago

Thank you for replying.

I am aware of negative sampling they used. It's more like an engineering hack to speed up stuff.

They also used noise contrastive estimation as another loss function candidate.

But, I want to double confirm that ordinary softmax with full cross entropy is perfect valid in terms of computation correctness instead of efficiency.

^ | v • Reply • Share ›

Bob • a year ago

Nice article, very helpful , and waiting for your negative sample article.
My two cents, to help avoid potential confusion :
First, the CODE : https://github.com/tensorfl...

First, the CODE : https://github.com/tensorfl...

Note though word2vec looks like a THREE-layer (i.e., input, hidden, output) neural network, some implementation actually takes a form of kind of TWO-layer (i.e., hidden, output) neural network.
To illustrate:
A THREE layer network means :
input \times matrix_W1 --> activation(hidden, embedding) -- > times matrix W2 --> softmax --> Loss
A TWO layer network means :
activation(hidden, embedding) -- > times matrix W2 --> softmax --> Loss

How ? In the above code, they did not use Activation(matrix_W1 \times input) to generate a word embedding.
Instead, they simply use a random vector generator to generate a 300-by-1 vector and use it to represent a word. They generate 5M such vectors to represent 5M words as their embeddings, say their dictionary consists of 5M words.
in the training process, not just the W2 matrix weights are updated, but also "the EMBEDDINGS ARE UPDATED" in the back-propogation training process as well.
In this way, they trained a network where there is no matrix W1 that need to be updated in the training process.
It confused me a little bit at my first look at their code, when I was trying to find "two" matrices.

Sorry I had to use Capital letter as highlight to save reader's time. No offence.

2 • Reply • Share ›

Chris McCormick Mod → Bob • a year ago

FYI, I've written a part 2 covering negative sampling.

1 • Reply • Share ›

Chris McCormick Mod → Bob • a year ago

I could be wrong, but let me explain what I think you are seeing.

As I understand it, your diagram of a "3-layer network" is incorrect because it contains three weight matrices, which you've labeled W1, word embeddings, and W2. The correct model only contains two weight matrices--the word embeddings and the output weights.

Where I could see the *code* being confusing is in the input layer. In the mathematical formulation, the input vector is this giant one-hot vector with all zeros except at the position of the input word, and then this is multiplied against the word embeddings matrix. However, as I explained in the post, the effect of this multiplication step is simply to select the word vector for the input word. So in the actual code, it would be silly to actually generate this one-hot vector and multiply it against the word embeddings matrix--instead, you would just select the appropriate

row of the embeddings matrix.

Hope that helps!

1 ^ | v • Reply • Share ›

Sanjay Rakshit • 16 days ago

Hi. Thanks for the awesome article. I have a question. In the article you have said that the training samples are (the, quick), (the, brown) etc... In another section I understood that the training sample is a one-hot encoded vector. I can understand one-hot encoding for a single word. But how do you do it for a tuple like that?

1 ^ | v • Reply • Share ›

Ajay Prasadh • a year ago

Explicitly defining the fake task helps a lot in understanding it. Thanks for an awesome article !

1 ^ | v • Reply • Share ›

Chris McCormick Mod → Ajay Prasadh • a year ago

Glad it helped, thanks!

^ | v • Reply • Share ›

Nazar Dikhil → Chris McCormick • 10 months ago

Please
I want to do predicted a fuzzy time series (fuzzification by FC-mean)
By RBF neural network
But I'm having a problem with training, can you help me ???
Thanks

^ | v • Reply • Share ›

Ahmed ElFki • a year ago

Thank you for this article, however in the hidden layer part how did you choose the number of features for the hidden layer weight matrix ?

1 ^ | v • Reply • Share ›

Chris McCormick Mod → Ahmed ElFki • a year ago

Hi, Ahmed - 300 features is what Google used in training their model on the Google news dataset. 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).

^ | v • Reply • Share ›

Ahmed ElFki → Chris McCormick • a year ago

Hi Chris, first thank you for answering my question and second i have

another question which is related to the word representation. After initializing the Word2Vec method (in python) with the its corresponding parameters (such as the array of sentences extracted from documents, number of hidden layers, window, etc) then each word will be described by a set of values (negative and positive) in the range -1, 1 which is a vector depending on the numbers of features chosen previously. As i understood from gensim documentation all of the words representation can be extracted by means of the method save_word2vec_format(dest_file_name, binary=False) that can be fed later to another network. So can you confirm if what i understood if right or wrong ?

^ | v • Reply • Share ›

Chris McCormick Mod ➦ Ahmed ElFki • a year ago

I believe that's all correct! In gensim, you can look up the word vector for a word using, e.g., .

```
model = gensim.models.Word2Vec.load_word2vec_format('./model/G

model['hello']
Out[12]:
array([-0.05419922,  0.01708984, -0.00527954,  0.33203125, -0.
       -0.01397705, -0.15039062, -0.265625  ,  0.01647949,  0.

np.min(model['hello'])
Out[15]: -0.59375

np.max(model['hello'])
Out[16]: 0.50390625
```

^ | v • Reply • Share ›

Ahmed ElFki ➦ Chris McCormick • a year ago

Hello Chris, thank you for having devoted part of your time to read and answer my question because it allows me to confirm my doubts on this API (Word2Vec). Your answers as well as your article help me a lot to progress in my project. :)

^ | v • Reply • Share ›

Chris McCormick Mod ➦ Ahmed ElFki • a year ago

Awesome, glad I could help!

^ | v • Reply • Share ›

Calvin Ku • a year ago

Thanks for the article Chris! I was going through a TensorFlow tutorial on Word2Vec and really couldn't make heads or tails of it. This article really helps a lot!

I have one question regarding the labels though. In the first figure, my understanding is, for each word (one-hot encoded vector) in the input, the NN outputs a vector of the same dimension (in this case, dim = 10,000) in which each index contains the probability of the

word of that index appearing near the input word. And since this is a supervised learning, we should have readied the labels generated from our training text, right (we already know all the probabilities from training set)? This means the labels are a vector of probabilities, and not a word, which doesn't seem to be agreed by your answer to **@Mostaphe**.

Also I don't think the probabilities in the output vector should sum up to one. Because we have a window of size 10 and in the extreme case, say we have a text of repeating the same sentence of three words over and over, then all the words will appear in the vicinity of any other word and they should always have probability of 1 in any case. Does this make sense?

1 ^ | ˅ • Reply • Share ›

Chris McCormick Mod ➦ Calvin Ku • a year ago

Hi Calvin, thanks, glad it was helpful!

The outputs of the Softmax layer are guaranteed to sum to one because of the equation for the output values--each output value is divided by the sum of all output values. That is, the output layer is normalized.

I get what you are saying, though, and it's a good point--I believe the problem is in my explanation.

Here is, I think, the more technically correct explanation: Let's say you take all the words within the window around the input word, and then pick one of them at random. The output values represent, for each word, the probability that the word you picked is that word.

Here's an example. Let's say in our training corpus, *every 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 words in the vicinity of 'York' and randomly pick one of them, the probability of it being 'New' *is not* 100%.

I will add a note to my explanation; thanks for catching this!

^ | ˅ • Reply • Share ›

Lifemakers Studio ➦ Chris McCormick • a year ago

So, for each input word, the ideal output of trained network should be a vector of 10,000 floating-point values, all of which should be 0 except those whose words have been ever found nearby the input word, and each such non-zero value should be proportional (pre-softmax) to the number of occurrences of that word near the input word?

If so, how can this ideal training be achieved if the network looks at only one nearby word at a time? For example if "York" has been seen 100 times near "new" and 1 time near "kangaroo", and we give the network the

York/kangaroo pair, wouldn't the training algorithm hike the output for "kangaroo" all the way to 1 at this step, instead of the 1/100 as it should be? Or does the fact that we'll feed it York/new pairs 100 times as often take care of this?

^ | v • Reply • Share ›

Chris McCormick Mod → Lifemakers Studio • a year ago

Your very last statement is correct. Each training sample is going to tweak the weights a little bit to more accurately match the output suggested by that sample. There will be many more samples of the "york" and "new" combination, so that pairing will get to tweak the weights more times, resulting in a higher output value for 'new'.

^ | v • Reply • Share ›

Sabih • 13 days ago

Thanks Chris for a great article. I have a question regarding prediction of context word in skip-gram model.

As per your example, lets take "fox" as the input word then, we have 4 training samples with window size 2:
(fox, quick)
(fox, brown)
(fox, jumps)
(fox, over)

Is the skip-gram going to predict only 1 of the 4 context words from "quick", "brown", "jumps", "over".

This confusion comes from this sentence: "Given a specific word in the middle of a sentence (the input word), look at the words nearby and pick one at random."

1) If I understand correct from your article then using "fox" as input, it can randomly select any of the 4 words from the sample and predict its 1-hot vector / probability distribution at output layer.

see more

^ | v • Reply • Share ›

DM SCU • 15 days ago

Knowledge couldn't be sexier.

^ | v • Reply • Share ›

Daneel Olivaw • 22 days ago

Really great article! It beats all the youtube videos I watched.

^ | v • Reply • Share ›

nag sumanth • 25 days ago

Great Article !!

• Reply • Share ›

Gitesh Khanna • a month ago

What an amazing explanation. Thanks alot!

• Reply • Share ›

Akatsuki • a month ago

loved reading this !

• Reply • Share ›

Guy Moshkowich • a month ago

Chris, great tutorial, simple and clear -- thanks.
One question though regarding the training of the network:
you wrote "the training output is also a one-hot vector representing the output word. " - but as the network tries to predict context distributions - I would expect that the training output will be the vector distribution of the input word in the text corpus i.e., not a one-hot vector but one with the probabilities of the input word appear in the context of the other 10,000 words.
Can you please clarify this point for me?

• Reply • Share ›

Hassan Azzam • 2 months ago

Consider N = window size, V = vocab. size
Is it applicable to use a hidden-layer(has N neurons) and N output layers(each has V neurons)?
output layers here represent predicted words.

• Reply • Share ›

Sanjana Khot • 2 months ago

Excellent explanation of something that seemed very difficult to understand when reading papers!

• Reply • Share ›

Allen Ji • 2 months ago

Thank you so much for the tutorial. Finally feel that I understand this:)

• Reply • Share ›

Shantanu Tripathi • 2 months ago

Very Nice intutive tutorial on word2vec! Thumbs up.

• Reply • Share ›

Dhairya Verma • 2 months ago

Excellent tutorial

^ | ˅ • Reply • Share ›

Ryan Rosario (DataJunkie) • 2 months ago

What software do you use to make the diagrams in your post? They are very clean!

^ | ˅ • Reply • Share ›

Elie • 2 months ago

Hi,
Thank you for the superb explanation.
Do you permit redrawing your figures and including them in my thesis while citing your work and giving you credit?

^ | ˅ • Reply • Share ›

Parth Vadhadiya • 2 months ago

awsome explaination.

^ | ˅ • Reply • Share ›

jasminyas • 2 months ago

This explanation was really fluid and simple ! love it and thank you for your work on putting all that together and explaining it so well ! :)

^ | ˅ • Reply • Share ›

Konstantin Kharlamov • 2 months ago

Thanks again, you described well how to use a ready word2vec ANN, but what about training algorithm?

UPD: also, regarding the usage: I think the novel idea of word2vec is about using geometric functions to measure distance in vector space between words, you somehow omitted it.

^ | ˅ • Reply • Share ›

Konstantin Kharlamov • 2 months ago

Thanks for tutorial and taking the time to figure how it works, I look forward to part 2 (which I'm going to read). I tried before that modifying the original word2vec code to figure out how does it work, but turned out it is absolutely awful — global variables, variables declared out of place where they used, and this line: "long long words, size, a, b, c, d, cn, bi[100];". Jesus Christ… I tried modifying ReadWord() to read words as well as symbols, and though the function works fine, but the whole thing just stopped working, and I suspect it broke yet another undocumented assumption somewhere.

It is terrible, there's no way to understand how it works other than rewriting from scratch.

^ | ⌄ • Reply • Share ›

Calvin Chan • 2 months ago

Thanks Christ, this is great, truly appreciated.

^ | ⌄ • Reply • Share ›

Tank S. • 2 months ago

I still don't quite understand how the Weight matrix is built? Is the weight trained from the input document (sentences) entirely, or just the word pairs?

^ | ⌄ • Reply • Share ›

Francisco Escolano • 3 months ago

Very nice tutorial! Congrats

^ | ⌄ • Reply • Share ›

Load more comments

ALSO ON **MCCORMICKML.COM**

K-Fold Cross-Validation, With MATLAB Code

1 comment • 2 years ago

Radha Kumari , IDD, Bioengg. & — It was useful for me, Thank you.

Gradient Descent Derivation

46 comments • 2 years ago

Tushar Sarde — Thanks!

Word2Vec Tutorial Part 2 - Negative Sampling

79 comments • a year ago

Leland Milton Drake — Hey Chris,When you say that only 300 weights in the hidden layer are updated, are you assuming that the …

AdaBoost Tutorial

15 comments • 2 years ago

Huey Kwik — This is really well-explained, thank you!I thought showing graphs of alpha vs. error and exp(x) vs. x was especially …

Subscribe Add Disqus to your siteAdd DisqusAdd Privacy

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

© 2017. All rights reserved.