

Word2Vec Tutorial - The Skip-Gram Model

19 Apr 2016

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

The Model

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

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

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

The Fake Task

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

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

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

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

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

Source Text	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)

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

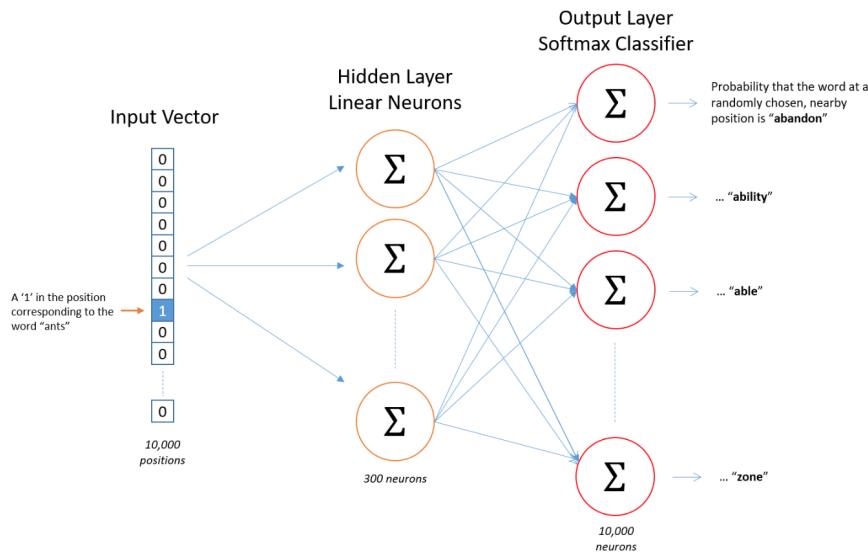
So how is this all represented?

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) and we'll place a "1" in the position corresponding to the word "ants", and 0s in all of the other positions.

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

Here's the architecture of our neural network.



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

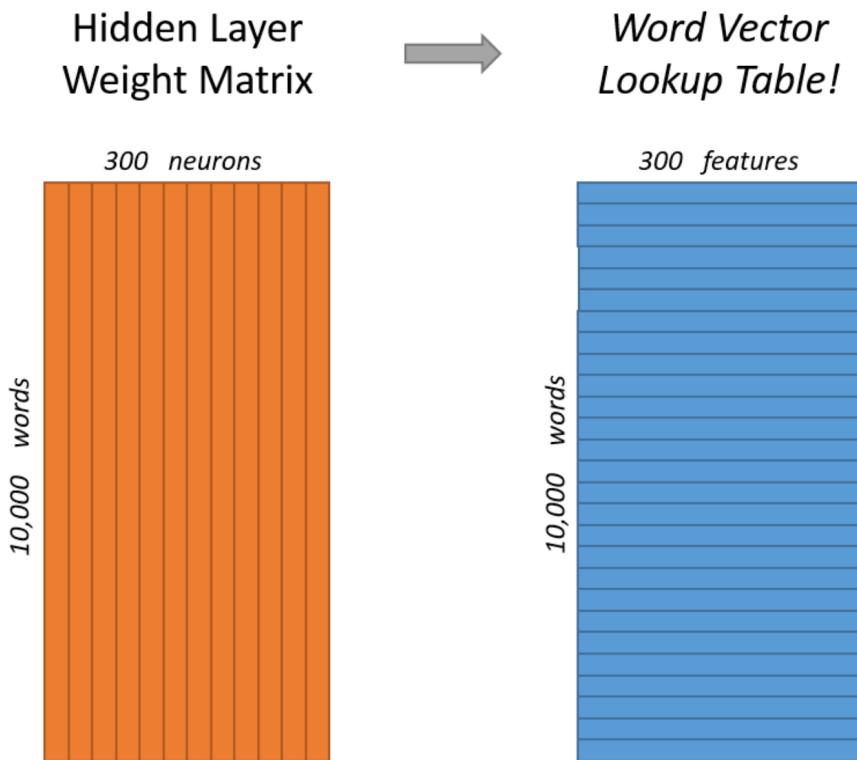
When *training* this network on word pairs, the input is a one-hot vector representing the input word and the training output is *also* a one-hot vector representing the output word. But when you evaluate the trained network on an input word, the output vector will actually be a probability distribution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

The Hidden Layer

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

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

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



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

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

Now, you might be asking yourself—"That one-hot vector is almost all zeros... what's the effect of that?" If you multiply a $1 \times 10,000$ one-hot vector by a $10,000 \times 300$ matrix, it will effectively just select the matrix row corresponding to the "1". Here's a small example to give you a visual.

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

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.

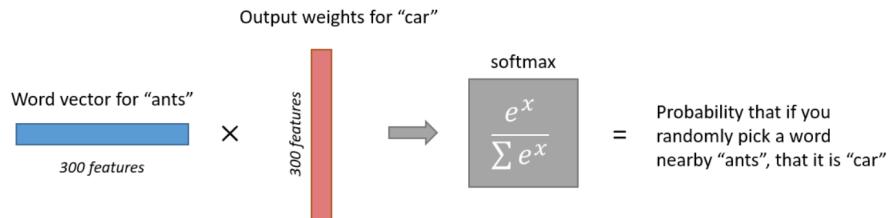
The Output Layer

The 1×300 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

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

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

Cite

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

▷ X

View PDF | Download Now

Free PDF File Reader - ViewPDF Extension. View PDF Files Instantly!

ViewPDF.io
OPEN

270 Comments [mccormickml.com](#) 1 Login ▾

Heart Recommend 182 Tweet Share
Sort by Best ▾

Join the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS ?

D
f
t
G
Name

raj1514 • 3 years ago • edited
Thanks for this post! It really saved time in going through papers about this...

75 ▲ | ▼ • Reply • Share ▾

Chris McCormick Mod ➔ raj1514 • 3 years ago
Great! Glad it helped.
1 ▲ | ▼ • Reply • Share ▾

zul waker • 3 years ago
You are amazing. I tried to understand this from so many sources but you gave the best

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

33 ^ | v · Reply · Share ·

 **Chris McCormick** Mod → zul waker · 3 years ago

Thanks so much, really glad it helped!

6 ^ | v · Reply · Share ·

 **micsca** · 3 years ago

nice article!

25 ^ | v · Reply · Share ·

 **ningyuwhut** · 2 years 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

20 ^ | v 1 · Reply · Share ·

 **Supun Abeysinghe** → ningyuwhut · 2 years 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 :)

10 ^ | v · Reply · Share ·

 **Chris McCormick** Mod → Supun Abeysinghe · 6 months ago

I concur!

1 ^ | v · Reply · Share ·

 **Anne Puzon** → ningyuwhut · a year ago

It's because you use the surrounding context of the current token (SKIPing the current token). It's a play on n-gram -- you preserve the n-gram "ordering" of the entire phrase (context words + current word) using weighting of the context words based on distance to the skipped word.

It's different than continuous bag of words (CBOW) which doesn't preserve the order because of the weighting of context words based on distance to current word.

2 ^ | v · Reply · Share ·

 **Arish Ali** · 3 years ago

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

7 ^ | v · Reply · Share ·

 **Chris McCormick** Mod → Arish Ali · 3 years ago

Awesome, thanks!

^ | v · Reply · Share ·

 **Mike Stopa** · a year ago

This is a great post. Very useful and pitched at a perfect level (for me, anyway). Thanks!

6 ^ | v · Reply · Share ·

 **Chris McCormick** Mod → Mike Stopa · 6 months ago

Thanks, Mike! I started my career as a software engineer and only later moved into machine learning, so I try to retain some of that "outsider" perspective in my explanations.

1 ^ | v · Reply · Share ·

 **Albert Wang** · 3 years ago · edited

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?

6 ^ | v 1 · Reply · Share



Chris McCormick Mod → Albert Wang · 3 years 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 your 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



Andres Suarez → Chris McCormick · a year ago

Hi Albert, Chris.

The position of a word within a given size window does affect the training, but indirectly. As explained in their paper (section 3.2 in <https://arxiv.org/pdf/1301....>), SkipGram reduces the sampling window randomly for every training word, to "give less weight to the distant words by sampling less from those words in our training examples".

BTW Albert, great posts, thanks a lot!

^ | v · Reply · Share



Albert Wang → Chris McCormick · 3 years ago · edited

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



Anne Puzon → Albert Wang · a year ago

Yes, it's valid. Mathematically, negative sampling loss is derived from a full cross entropy (after a few simplifying assumptions are made)

1 ^ | v · Reply · Share



Bob · 3 years ago · edited

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...>

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-propagation 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 ^ | v · Reply · Share



Chris McCormick Mod → Bob · 3 years ago

FYI, I've written a [part 2](#) covering negative sampling.

1 ^ | v · Reply · Share



Chris McCormick Mod → Bob · 3 years 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 ⤵ · Reply · Share ↗



yongle · 8 months ago

like the intuition part!

1 ⤵ · Reply · Share ↗



Janothon · a year ago

Thank you for your crisp explanation.

I have one question though concerning the output layer of the skip-gram model: I often see figures that look like there would be c different output distributions where c is the window size. What is the trick here? Is the softmax function different for every context word? Thank you for your help.

1 ⤵ · Reply · Share ↗



Sanjay Rakshit · 2 years 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 ⤵ · Reply · Share ↗



Songtao Lin → Sanjay Rakshit · 2 years ago

Hi Sanjay! I am pretty new to this so I maybe wrong. In my understanding, the tuple just means one input(x) and one output(y). So you may have the one hot encoding for "the" in one row of your X_train variable and one hot encoding for "quick" in the corresponding row of your y_train variable. And you do this for every tuple in your training set.

⤵ · Reply · Share ↗



Ajay Prasad · 3 years ago · edited

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

1 ⤵ · Reply · Share ↗



Chris McCormick Mod → Ajay Prasad · 3 years ago

Glad it helped, thanks!

⤵ · Reply · Share ↗



Nazar Dikhil → Chris McCormick · 2 years 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

⤵ · Reply · Share ↗



Ahmed ElFki · 3 years 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 ⤵ · Reply · Share ↗



Chris McCormick Mod → Ahmed ElFki · 3 years 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).

⤵ · Reply · Share ↗



Ahmed EIFki → Chris McCormick · 3 years 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 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 is right or wrong ?

^ | v • Reply • Share ›



Chris McCormick Mod → Ahmed EIFki · 3 years ago · edited

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/GoogleNews-vectors-negative300.bin', binary=True)
model['hello']
Out[12]: array([-0.05110022, -0.01700001, -0.00577051, 0.22702125, -0.25,
```

```
[-0.01397705, -0.15039062, -0.265625, 0.01647949, 0.3828
np.min(model['hello'])
Out[15]: -0.59375
np.max(model['hello'])
Out[16]: 0.50390625
```

^ | v · Reply · Share ·



Ahmed EIFki → Chris McCormick · 3 years 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 EIFki · 3 years ago

Awesome, glad I could help!

^ | v · Reply · Share ·



Calvin Ku · 3 years 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 ^ | v · Reply · Share ·



Chris McCormick Mod → Calvin Ku · 3 years ago · edited

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!

^ | v · Reply · Share ·



Lifemakers Studio → Chris McCormick · 3 years 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 · 3 years ago

[Chris McCormick](#) · [Architectural Details](#) · [Your Posts](#)

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](#) ·



Unknown · 18 days ago

In which language you code the tutorial for word2vec in your book ?

[^](#) | [v](#) · [Reply](#) · [Share](#) ·



Suhas Shrinivasan · a month ago

Really appreciate how this article gets to the point, into the technical details, that I was more interested in knowing, and then conveys the intuition. There were too many blogposts that simply conveyed the intuition abstractly, with lots of examples, but barely ever got to how the neural network is trained and used.

[^](#) | [v](#) · [Reply](#) · [Share](#) ·



Jonathan Cui · 2 months ago

So is there bias for the hidden layer and the output layer?

[^](#) | [v](#) · [Reply](#) · [Share](#) ·



markusloecher · 5 months ago · edited

Indeed, extremely well written, thanks a lot ! I am wondering if (and if not why not) Google and others also provide the "right side" of the hidden layer, that is, the "output weights". That matrix would be the same size and could be extremely useful for further research and necessary if one wanted to compute the actual probabilities of other words being present within a window

[^](#) | [v](#) · [Reply](#) · [Share](#) ·



Arjun · 5 months ago

Very nicely explained. But I have a doubt, as in the example "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)" Couldn't understand this part. How can we represent a single hidden layer with 300 columns and 10000 rows?

[^](#) | [v](#) · [Reply](#) · [Share](#) ·



Nguyễn Thế Quý · 5 months ago

i would like to cite your post, is it possible ?

[^](#) | [v](#) · [Reply](#) · [Share](#) ·



Kyrill Alyoshin · 6 months ago

Chris, I think the purchasing site for your w2v book is broken. My purchase fails with "Missing Purchase" error, probably because the actual item that it lists is called "Dynamically Updated" and has a price of \$0. Somebody needs to look into it.

[^](#) | [v](#) · [Reply](#) · [Share](#) ·



Chris McCormick Mod → Kyrill Alyoshin · 6 months ago

Thanks Kyrill—we're looking into it!

[^](#) | [v](#) · [Reply](#) · [Share](#) ·



Sudhir Ghandikota · 7 months ago

Hello Chris,

Great article! Simple yet effective. I just had one question (or observation). Are both CBOW and Skip-gram models exactly the same if we use the window-size as 1 in the skip-gram model?

[^](#) | [v](#) · [Reply](#) · [Share](#) ·



Maggie Hu · 7 months ago

Thanks a lot. There I still get confused about the output size.

It's okay to split the word into pairs but the original pattern of skip-gram is to give 2*window size softmax classifications. So in order to train pairs data, need to keep pairs with the same center word/ focus word in the exact same batch when using SGD to optimize?

[^](#) | [v](#) · [Reply](#) · [Share](#) ·



Eric Kim · 7 months ago

Great article! very clear and easy to understand the dimension issues of the matrix calculation.

Thanks

[^](#) | [v](#) · [Reply](#) · [Share](#) ·



Chris McCormick Mod → Eric Kim · 5 months ago

Thanks Eric!

[Load more comments](#)

ALSO ON MCCORMICKML.COM

MinHash Tutorial with Python Code

23 comments • 3 years ago



ben — Converting and between binary and ascii sounds a lot like a hashing operation and apparently it can be used for that. However, it

Deep Learning Tutorial - PCA and Whitening

1 comment • 3 years ago



Jae Duk Seo — good post.

Image Derivative

16 comments • 3 years ago



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

XLNet Fine-Tuning Tutorial with PyTorch

6 comments • a month ago



John Fields — I had 2500 examples labeled as 1 and 350 examples labeled as 0. I thought that BERT would automatically scale the examples

 [Subscribe](#) [Add Disqus to your site](#) [Disqus' Privacy Policy](#)**DISQUS**

Related posts[XLNet Fine-Tuning Tutorial with PyTorch](#) 19 Sep 2019[BERT Fine-Tuning Tutorial with PyTorch](#) 22 Jul 2019[BERT Word Embeddings Tutorial](#) 14 May 2019

© 2019. All rights reserved.