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

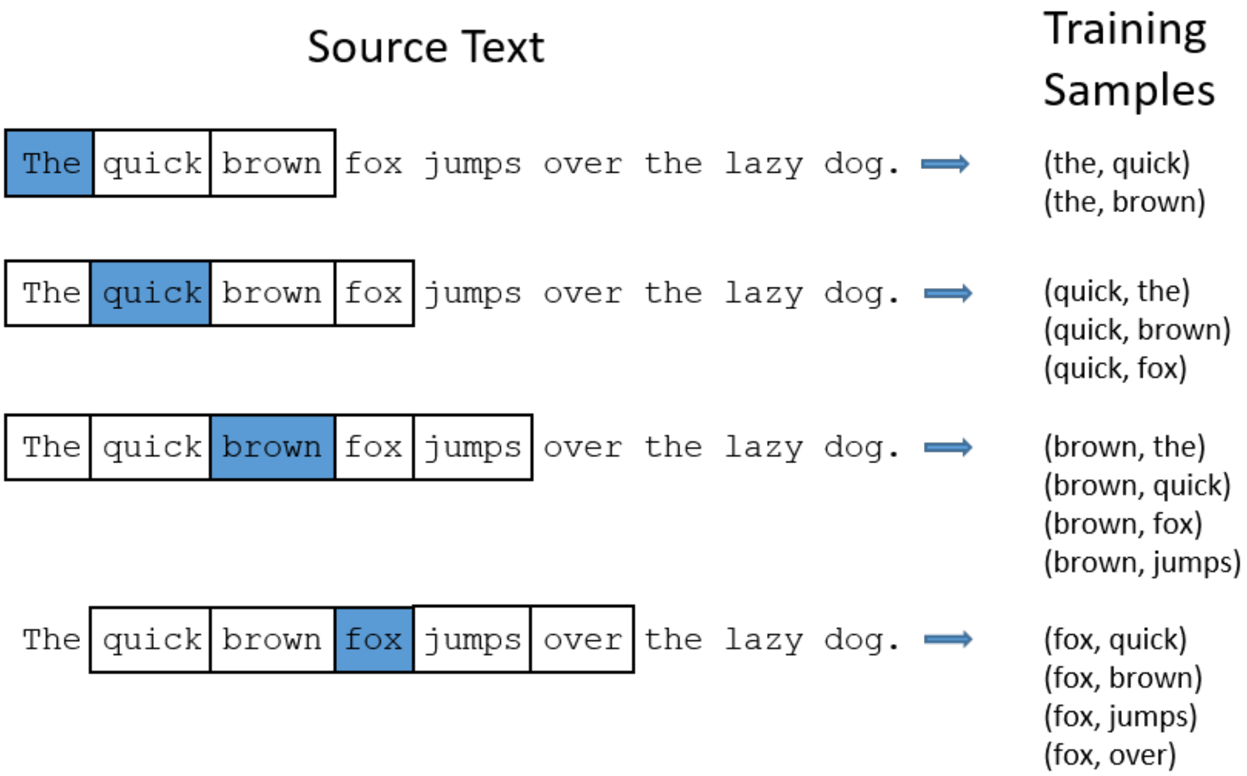
So now we need to talk about this “fake” task that we’re going to build the neural network to perform, and then we’ll come back later to how this indirectly gives us those word vectors that we are really after.

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

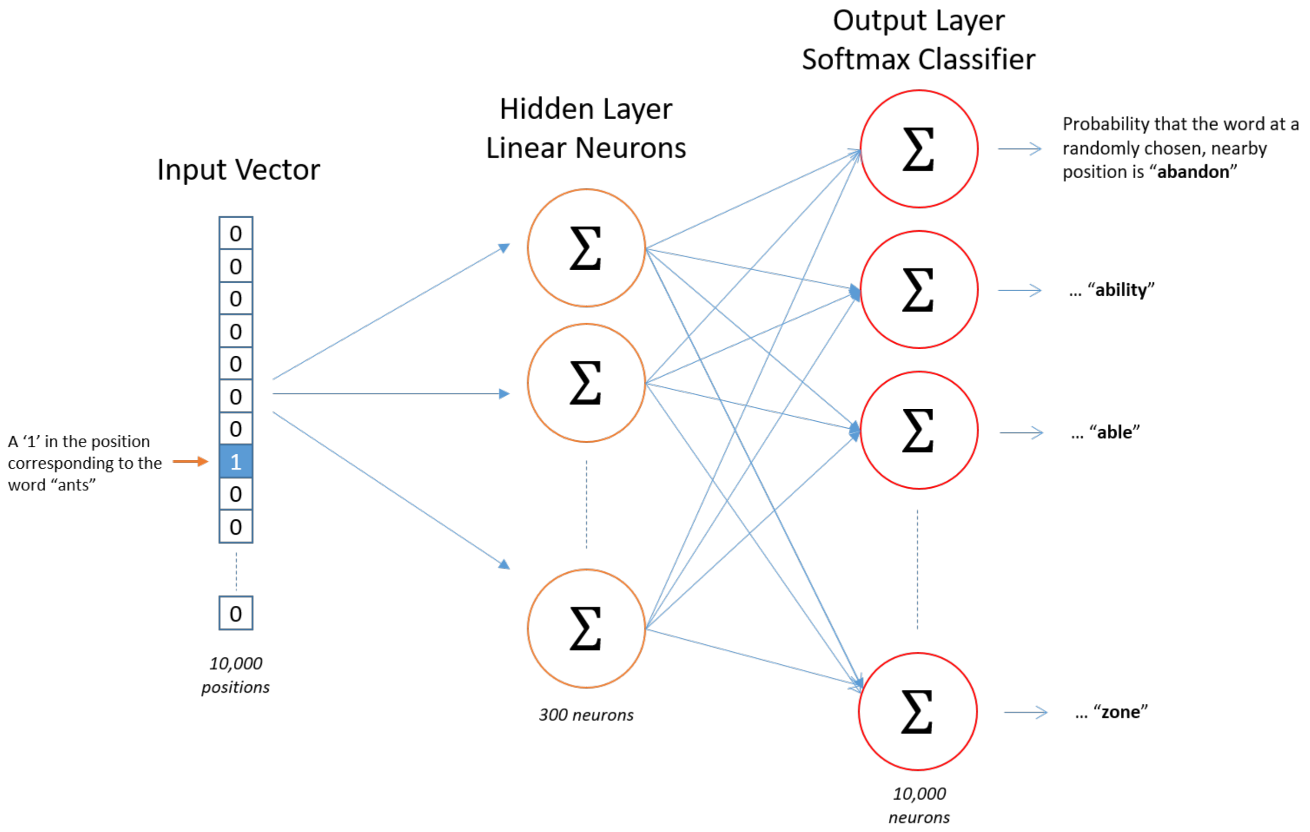
So how is this all represented?

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.



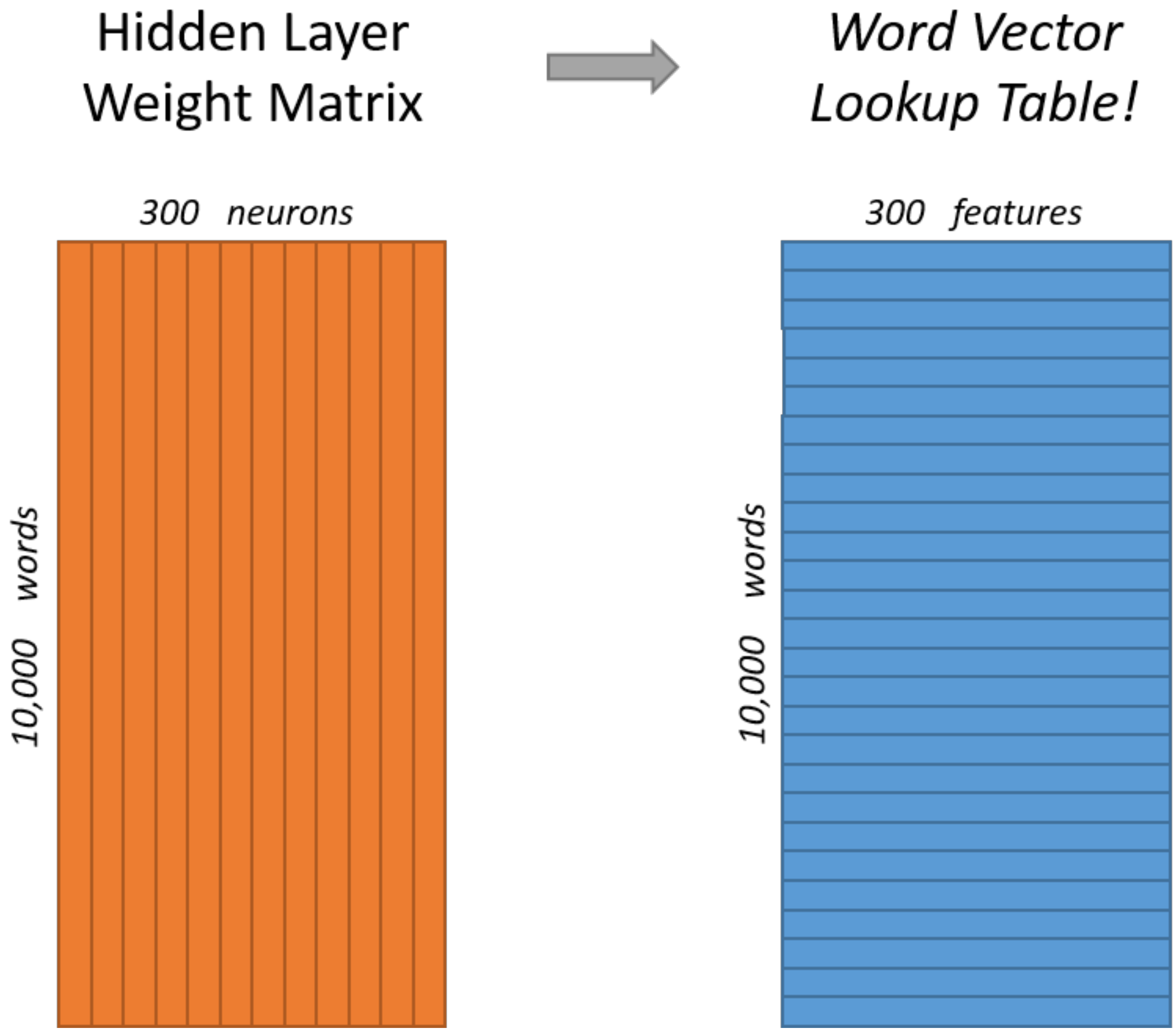
There is no activation function on the hidden layer neurons, but the output neurons use softmax. We'll come back to this later.

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).

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



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

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

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

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

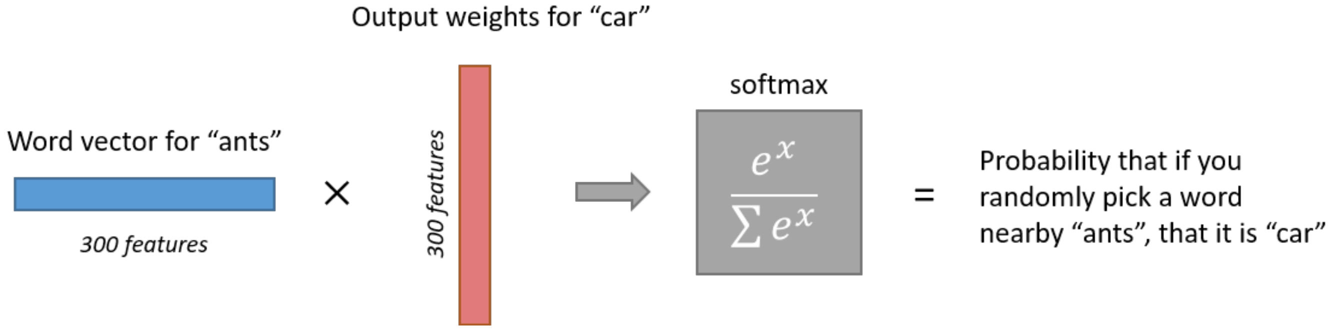
This means that the hidden layer of this model is really just operating as a lookup table. The output of the hidden layer is just the "word vector" for the input word.

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.

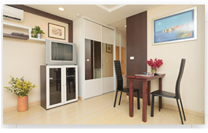
This can also handle stemming for you – the network will likely learn similar word vectors for the words "ant" and "ants" because these should have similar contexts.

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.



NEW DELHI
YOU BELONG HERE. Book now from ₹3300/night.
airbnb.co.in

36 Comments mccormickml.com 1 Login

Recommend 16 Share Sort by Best

Join the discussion…

Calvin Ku • 3 months 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 • 3 months 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 ›

Ravi Teja • 9 days ago

Great article Chris, I'm trying to do the udacity course on deep learning and now I have a better grasp of things.The one doubt I have is for example if I train my network with the input "The quick brown fox jumps over the lazy dog", the Following is going to be the input to the network right?

Feature, Target
the,quick
the,brown
quick,the
quick,brown
quick,fox
brown,the
brown,quick
brown,fox
brown,jumps
fox,quick
fox,brown
fox,jumps
fox,over

When I input the word, 'fox' into the network , it should give equal probabilities to 'quick','brown','jumps' and 'over' right ?

⌃ | ⌄ • Reply • Share ›

Chris McCormick Mod ➦ Ravi Teja • 7 days ago

Looks good to me! Of course, I wouldn't expect the neural network to *exactly* match the probabilities calculated in the training data, but you've got the right general idea.

⌃ | ⌄ • Reply • Share ›

Ravi Teja ➦ Chris McCormick • 7 days ago

Thank you Chris !

⌃ | ⌄ • Reply • Share ›

raj1514 • 13 days ago

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

⌃ | ⌄ • Reply • Share ›

Chris McCormick Mod ➦ raj1514 • 13 days ago

Great! Glad it helped.

⌃ | ⌄ • Reply • Share ›

Alexander Yau • a month ago

Nice article. Thank you!

⌃ | ⌄ • Reply • Share ›

Chris McCormick Mod ➦ Alexander Yau • a month ago

Thanks, Alexander!

⌃ | ⌄ • Reply • Share ›

Bob • 2 months 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/tensorflow/...

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.

⌃ | ⌄ • Reply • Share ›

Chris McCormick Mod ➦ Bob • 11 days ago

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

⌃ | ⌄ • Reply • Share ›

Chris McCormick Mod ➦ Bob • a month 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!

⌃ | ⌄ • Reply • Share ›

Labiba Jahan • 2 months ago

Thanks for the nice article. I just want to be sure that my understanding is correct.
Suppose I want to implement word2vec on 10 words with 5 features . Then I want to know that the following steps are correct or not.
1.convert each word with boolean matrix. dimension: 10*10
2. Multiply this matrix with hidden layer matrix which contains random weights. (10*10 multiply with 10*5)
3. Multiply each word vector with the whole matrix. (1*10 multiply with 10*5) = x
4. Calculate the value by e^x/sum of(e^x)
Now my question is, how to calculate x? In my sense it is a vector. how can I convert it to one single variable?

⌃ | ⌄ • Reply • Share ›

Chris McCormick Mod ➦ Labiba Jahan • a month ago

Thanks, Labiba!

Let's see if I can help...

1. Input should be a one-hot vector, 1x10
2. Multiply this by the hidden layer matrix, 10x5 (1x10 * 10x5 = 1x5)
3. Multiply the output of the hidden layer by the output weights, 5x10 (1x5 * 5x10 = 1x10)

"x" is the 1x10 vector output by step 3.

For each component in "x", calculate exp(x). Then divide each component by sum(exp(x)). So your final output is still a 1x10 vector.

Hope that helps!

^ | ˅ • Reply • Share ›

Labiba Jahan ➦ Chris McCormick • 25 days ago

Wow! got it now. Thanks.

^ | ˅ • Reply • Share ›

Karthik Suresh • 2 months ago

Great Article!! It would be even more helpful if you had thrown some light on Negative Sampling is well.

^ | ˅ • Reply • Share ›

Chris McCormick Mod ➦ Karthik Suresh • 11 days ago

FYI, I've posted a part 2 that covers Negative Sampling.

^ | ˅ • Reply • Share ›

Chris McCormick Mod ➦ Karthik Suresh • 2 months ago

Thanks! Yes, I'm still hoping to come back and write another post on the enhancements they made like Negative Sampling--I'll have to do some work to understand it myself, first, though!

^ | ˅ • Reply • Share ›

Homayun • 3 months ago

Wowww, what a great explanation. Very clear and understandable. Just a minor comment. I would rather say "the probability is 1" instead of "the probability is 100%" ;) BTW, thanks a lot for the nice tutorial.

^ | ˅ • Reply • Share ›

Chris McCormick Mod ➦ Homayun • 3 months ago

Awesome, glad it was helpful!

^ | ˅ • Reply • Share ›

Mostaphe • 4 months ago

Nice article, I have a question to ask please, it is supervised or unsupervised learning? in case it is a supervised what is the expected output?

^ | ˅ • Reply • Share ›

Chris McCormick Mod ➦ Mostaphe • 3 months ago

It's supervised learning. Each training example is simply a pair of words--an input word and an output word, both represented as one-hot vectors (A vector with one position for every word in the vocabulary, and all set to zero except for one component). The output word is just a word that was found in the vicinity of the input word.

When the training is all complete, if you feed the network an input word, what you will get on the output is a vector of fractions. Again, there is one position for each word in the vocabulary, and the value at a given position reflects how likely it is that that word will appear in the vicinity of the input word.

^ | ˅ • Reply • Share ›

Vincent → Chris McCormick • 15 days ago

Thanks for this great article. I have a question about the output. So the output is a vector of fractions (probabilities) where each number represents how likely a word will appear in the vicinity.

Should the fractions (probabilities) sum up to be 1? Suppose there are two words in vicinity of the input word definitely appear together with the input word, doesn't the fractions (probabilities) of these two words should both be 1? So will the sum of fractions of the output can be much greater than 1?

^ | ˅ • Reply • Share ›

Chris McCormick Mod → Vincent • 14 days ago

The outputs will all sum up to 1.

Your question is valid--I need to go back and fix the wording in my post to clarify this better. Check out my side note where I say "the correct interpretation of the outputs of the model is slightly different from what I've said".

The probability for a single output word will never be 1 because you don't which of the nearby words you've picked. Even if "New" *always* appears next to "York", the probability is less than 1 because you don't know whether you've picked the word before York or the word after, or two words before, etc.

^ | ˅ • Reply • Share ›

Rayees Dar • 4 months ago

Great article. Helped to understand the algorithm particularly explaining with the "fake" trick.

Just a question here (might look silly).
I was looking at word2vec tutorials, how to use them. Couldn't exactly figure out how. I just want to use the dense vector representation of the words for use in RNN for a different task (I am building a summarizer). Can you explain how to go about it. Should I get the representations by training on my own data or the trained model will do. More importantly please tell me how to get them (using pre-trained models using

gensim)

• Reply • Share ›

Chris McCormick Mod ➦ Rayees Dar • 3 months ago

I'm not experienced enough with the model to tell you whether to use a pre-trained one or train your own--I suppose the reason to train your own would be if you think that a lot of the word meanings are specific to the context of your application.

If you want to use a pre-trained one, though, check out my post on using Google's model with gensim. I created a small project on GitHub that does exactly that called inspect_word2vec.

• Reply • Share ›

Mahdi Dibaiee • 4 months ago

Great post, helped me a lot in understanding the skip-gram model. Thank you!

• Reply • Share ›

Chris McCormick Mod ➦ Mahdi Dibaiee • 4 months ago

Awesome, thanks!

• Reply • Share ›

leegongzi • 4 months ago

I have a question, when training this network, what's the output?I mean the 'label' of the input. Is it the one-hot vector of a word which appeared with the input word in a window. if it means that we should write a program to count them?

• Reply • Share ›

Chris McCormick Mod ➦ leegongzi • 4 months ago

You've got it. The training samples are actually just word pairs. That is, a single training sample consists of the input word and one other word that was found nearby. And both the input and output are represented by one-hot vectors.

If you look at the implementation, you'll find that there are some extra tricks they get into where they sample the words--they don't train on every single word pair. I'm hoping I'll get to share an explanation of that technique on my blog eventually (I don't entirely understand it myself yet).

• Reply • Share ›

leegongzi ➦ Chris McCormick • 4 months ago

Thank for your help, I've got it. expect your next blog..

• Reply • Share ›

ray • 5 months ago

great tutorial that makes me clear how word2vec works internally. Thanks!

⌃ | ⌄ • Reply • Share ›

Chris McCormick Mod ➦ ray • 5 months ago

Glad it was helpful, thanks!

⌃ | ⌄ • Reply • Share ›

Shayan Zamani • 7 months ago

thank you so so so much. This post solved many of my confusions. But I have a question: what exactly dimension is? I mean, you built this neural network with 300 neurons in the hidden layer (which is the dimension of our vector space) but why not let's say 3 dimensions? why not more that 301?, etc. Is there any logic behind of this number (300)? because I am a little confused about dimensions in this approach (word2vec), I know the dimensions in the tf-idf or co-occurrence approaches, each row is a word and each column is a word (or document) that occur around this word, but here what dimension is? and why 300?

⌃ | ⌄ • Reply • Share ›

Tomas Peterka ➦ Shayan Zamani • 6 months ago

The dimensionality depends on the problem you want to solve using word2vec. Let suppose we want to capture semantic relations in English. There is a great article about compressing word2vec which I unfortunately can't find right now. In short, the article claims that the amount of information captured by word2vec has a logarithmic shape. Than the article showed that bellow 100 dimensions the information was not really preserved. The curve was flattening all the way towards 300 dimension. In space greater than 300 dimensions your information get diluted and thus the arithmetics doesn't work as nicely as in more dense space.

⌃ | ⌄ • Reply • Share ›

micsca • 8 months ago

nice article!

⌃ | ⌄ • Reply • Share ›

ALSO ON **MCCORMICKML.COM**

Gaussian Mixture Models Tutorial and MATLAB Code

23 comments • 9 months ago•

chebbi safa — Hi Chris,Thank you very much for the very simple and interesting tutoriel Can you please mention a

Stereo Vision Tutorial - Part I

5 comments • 9 months ago•

Chris McCormick — Thanks, Beto, glad it was helpful!

tutorial.Can you please mention a …

Word2Vec Tutorial Part 2 - Negative Sampling

4 comments • 11 days ago•

Chris McCormick — Hi, Bob -Sorry that your comments were getting marked as spam--don't know what happened …

Deep Learning Tutorial - Sparse Autoencoder

3 comments • 9 months ago•

Choung young jae — For a given hidden node, it’s average activation value (over all the training samples) should be a small …

Subscribe Add Disqus to your site Add Disqus Add Privacy

Related posts

Word2Vec Tutorial Part 2 - Negative Sampling 11 Jan 2017

DBSCAN Clustering 08 Nov 2016

Interpreting LSI Document Similarity 04 Nov 2016

© 2017. All rights reserved.