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

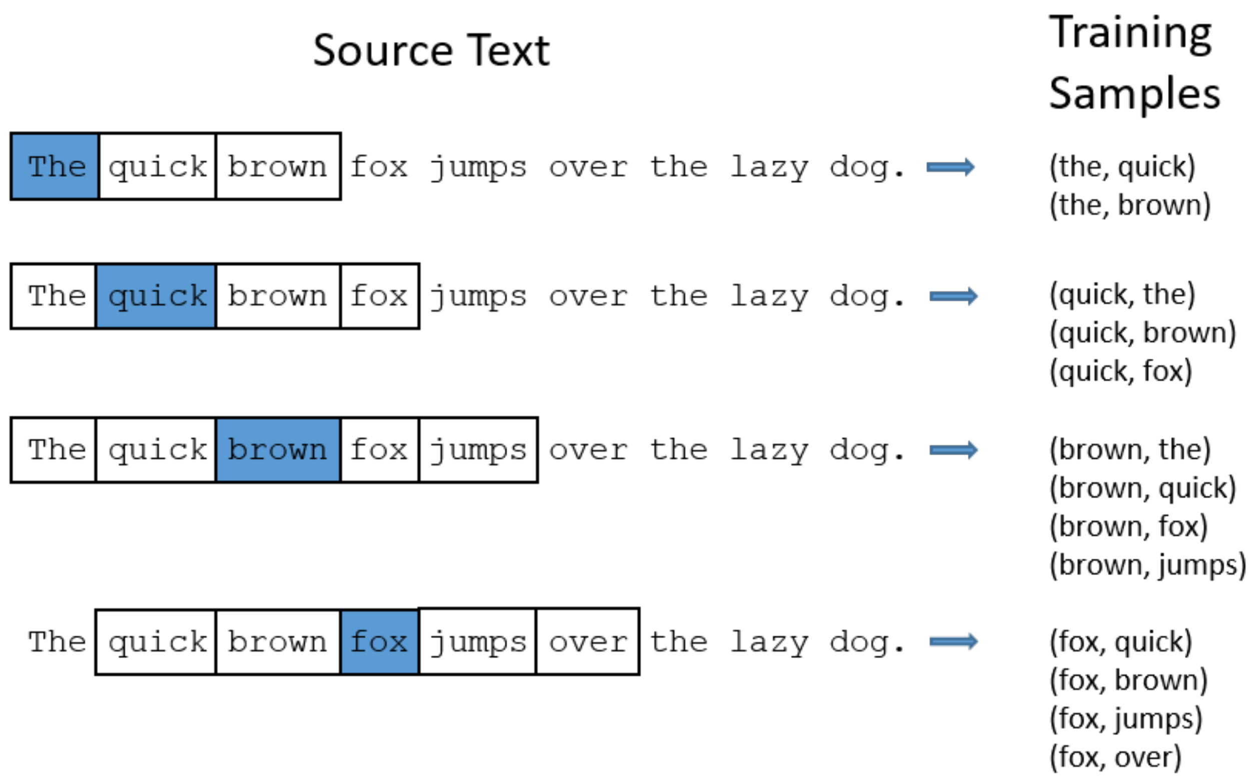
So now we need to talk about this “fake” task that we’re going to build the neural network to perform, and then we’ll come back later to how this indirectly gives us those word vectors that we are really after.

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

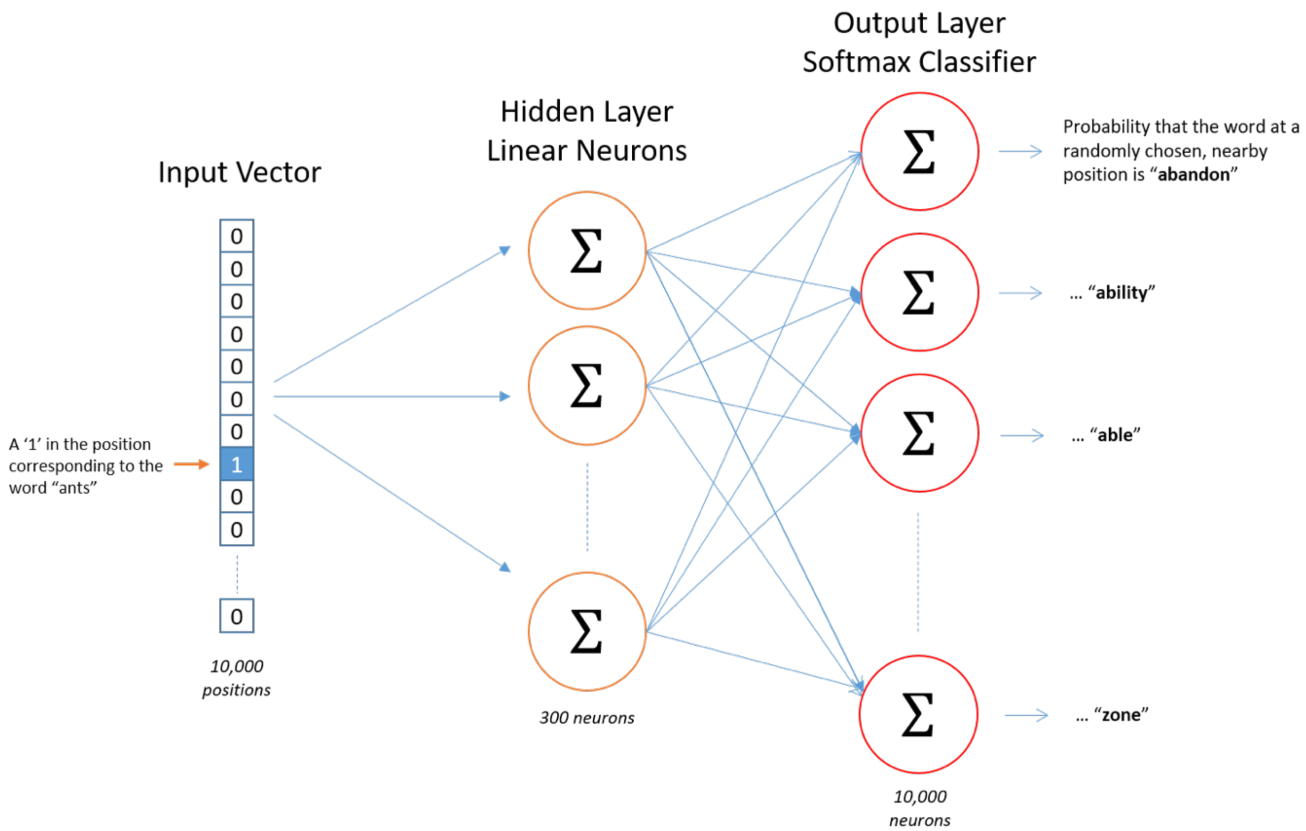
So how is this all represented?

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.

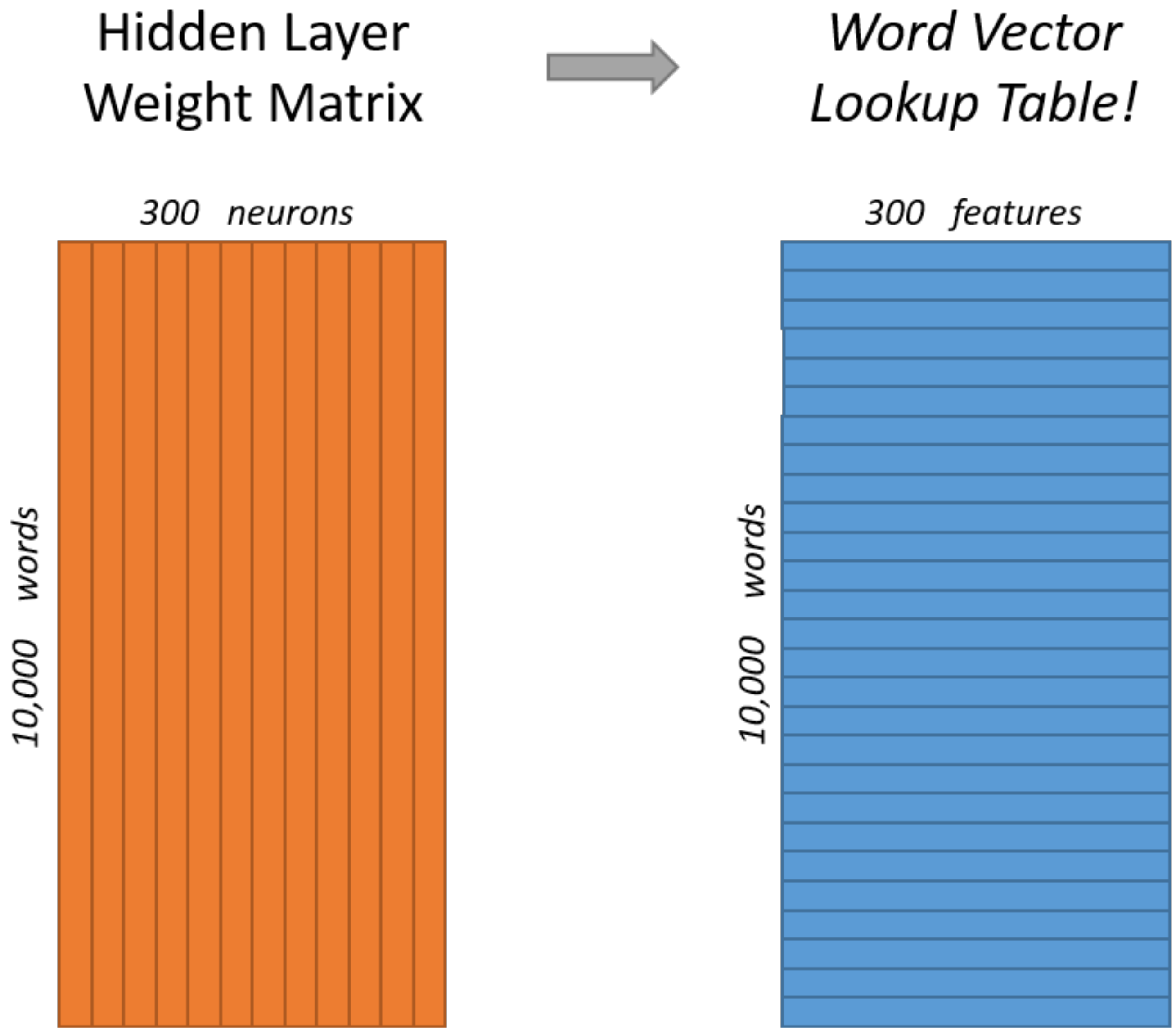
When *training* this network on word pairs, the input is a one-hot vector representing the input word and the training output *is also a one-hot vector* representing the output word. But when you evaluate the trained network on an input word, the output vector will actually be a probability distribution (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!



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

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

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

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

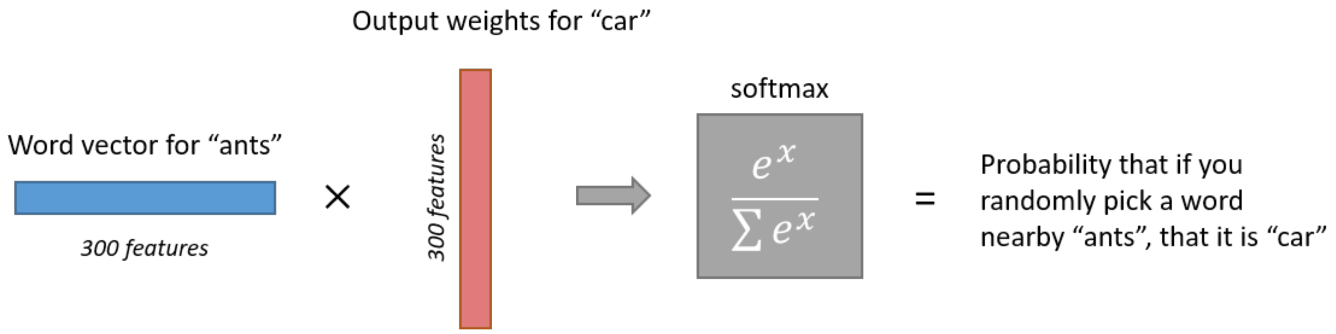
This means that the hidden layer of this model is really just operating as a lookup table. The output of the hidden layer is just the "word vector" for the input word.

The Output Layer

The `1 x 300` word vector for "ants" then gets fed to the output layer. The output layer is a softmax regression classifier. There's an in-depth tutorial on Softmax Regression here, but the gist of it is that each output neuron (one per word in our vocabulary!) will produce an output between 0 and 1, and the sum of all these output values will add up to 1.

Specifically, each output neuron has a weight vector which it multiplies against the word vector from the hidden layer, then it applies the function `exp(x)` to the result. Finally, in order to get the outputs to sum up to 1, we divide this result by the sum of the results from *all* 10,000 output nodes.

Here's an illustration of calculating the output of the output neuron for the word "car".



> 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

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 ›

Chris McCormick Mod zul waker • a year ago

Thanks so much, really glad it helped!

4 • Reply • Share ›

raj1514 • a year ago

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

25 ^ | v • Reply • Share ›

Chris McCormick Mod → raj1514 • a year ago

Great! Glad it helped.

^ | v • Reply • Share ›

ningyuwhut • 7 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 • 7 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!

10 ^ | 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 ^ | ∨ • 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.

^ | ∨ • 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.

^ | ∨ • 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...

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 :

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 • Reply • Share ›

Sanjay Rakshit • 2 months 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 • 4 hours 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 Prasadh • a year ago

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

1 • Reply • Share ›

Chris McCormick Mod → Ajay Prasadh • a year ago

Glad it helped, thanks!

• Reply • Share ›

Nazar Dikhil → Chris McCormick • a year 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 • 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 • 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).

• 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 ?

^ | ˅ • 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('./mo

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

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

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

^ | ˅ • 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. :)

^ | ˅ • Reply • Share ›

Chris McCormick Mod ➦ Ahmed ElFki • a year ago

Awesome, glad I could help!

^ | ˅ • 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

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 • 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!

^ | v • 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'.

• Reply • Share ›

Andy Wang • 10 days ago

Thanks for this great article. Have a question, what is the shape of weight at the hidden to output layer ("output weight for car") ? It seems to me that it is 300 x 1? If this is the case, wouldn't it yield a shape of 1x1 output layer for every word? thanks

• Reply • Share ›

Owen Chapman • a month ago

Great blog. Curious about the properties of the output layer weight matrix; if the hidden layer matrix is our word representations, then how does this [vocab_size x embedding_dim] vector relate to the [embedding_dim x vocab_size] output weight matrix? could we use the transpose of these output in the same way as a dense word vector?

• Reply • Share ›

Ciprian-Bogdan Chirila • a month ago

Very nice and accessible explanations for the given example! Congratulations!

• Reply • Share ›

GeorgeRififi • a month ago

thanks man!

• Reply • Share ›

Sabih • 2 months 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

• Reply • Share ›

DM SCU • 2 months ago

Knowledge couldn't be sexier.

• Reply • Share ›

Daneel Olivaw • 2 months ago

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

• Reply • Share ›

nag sumanth • 2 months ago

Great Article !!

• Reply • Share ›

Gitesh Khanna • 2 months ago

What an amazing explanation. Thanks alot!

• Reply • Share ›

Akatsuki • 2 months ago

loved reading this !

• Reply • Share ›

Guy Moshkowich • 2 months 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 • 3 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 • 3 months ago

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

Reply • Share ›

Allen Ji • 3 months ago

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

Reply • Share ›

Shantanu Tripathi • 3 months ago

Very Nice intutive tutorial on word2vec! Thumbs up.

Reply • Share ›

Dhairya Verma • 3 months ago

Excellent tutorial

Reply • Share ›

Ryan Rosario (DataJunkie) • 3 months ago

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

Reply • Share ›

Elie • 3 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 • 3 months ago

awsome explaination.

Reply • Share ›

jasminyas • 3 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 ›

Load more comments

some errors are to be expected, especially if there's not a lot of visual distinction in …

yoch melka — very great explanation, thank you !

The Gaussian Kernel

3 comments • 2 years ago

Ігор Яловецький — Ok, i like your explanation about Gaussian curve, but what about the kernel?

Radial Basis Function Network (RBFN) Tutorial

36 comments • 2 years ago

ايهاب ربايعة ابو مهند — amazingthanks for informative explanations

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