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

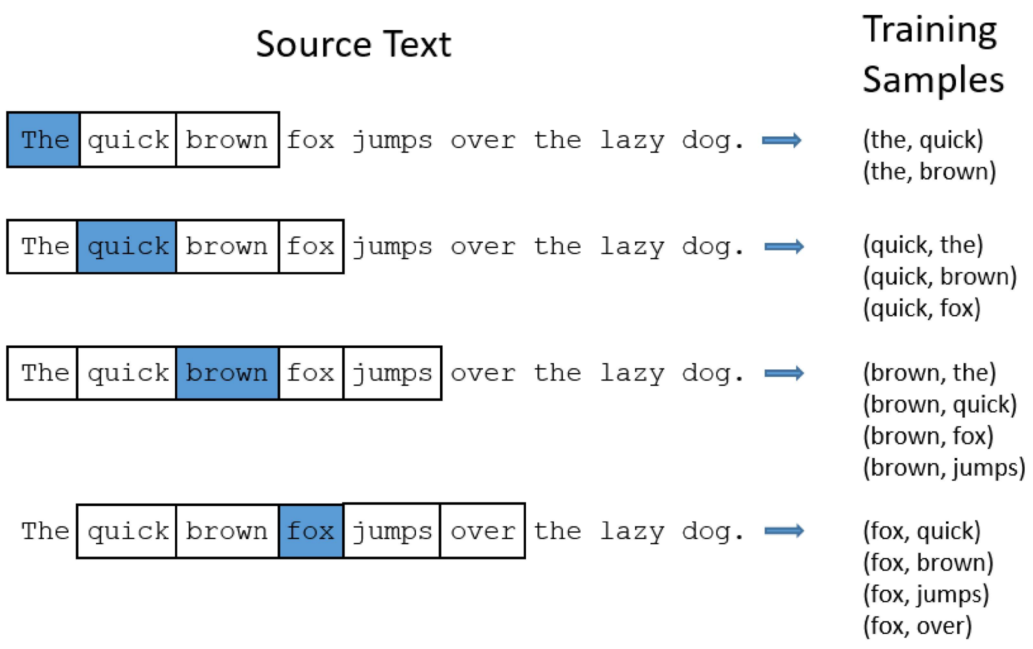
So now we need to talk about this "fake" task that we're going to build the neural network to perform, and then we'll come back later to how this indirectly gives us those word vectors that we are really after.

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.

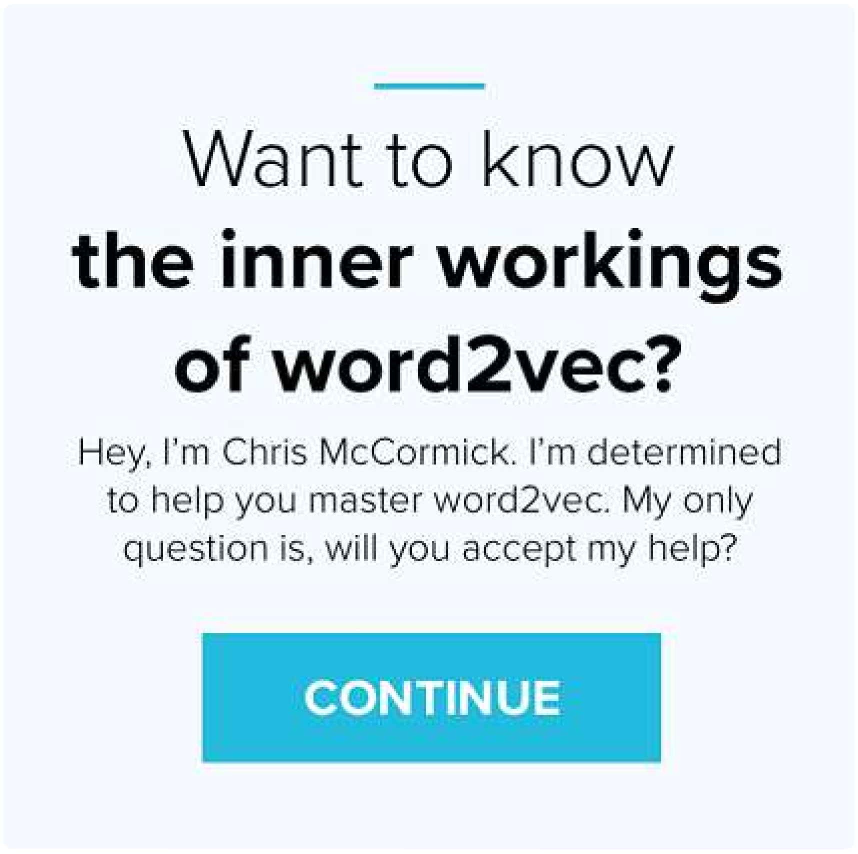
> When I say "nearby", there is actually a "window size" parameter to the algorithm. A typical window size might be 5, meaning 5 words behind and 5 words ahead (10 in total).

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.



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

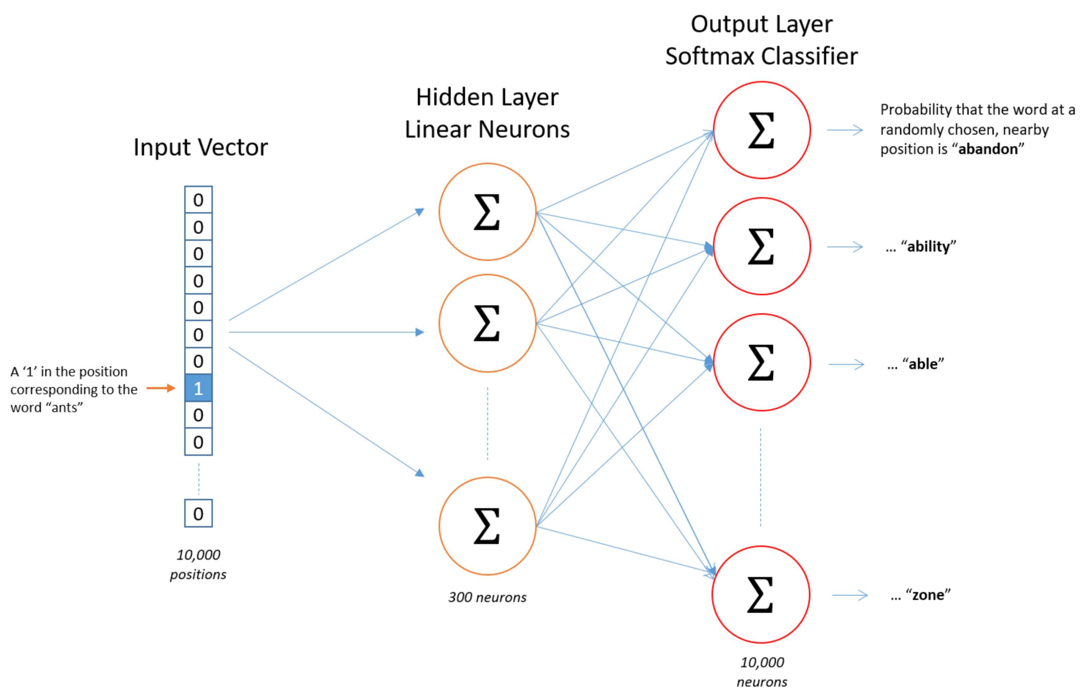
So how is this all represented?

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.

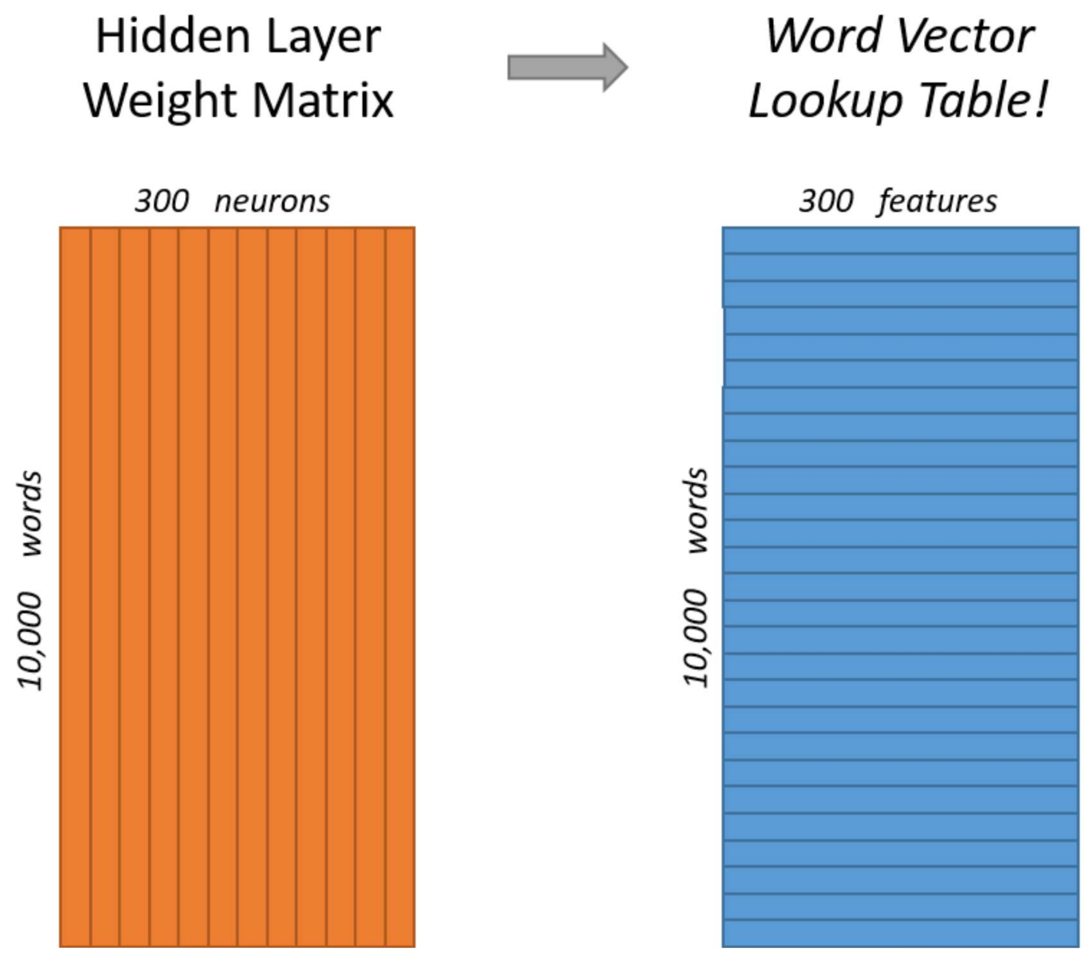
When *training* this network on word pairs, the input is a one-hot vector representing the input word and the training output *is also a one-hot vector* representing the output word. But when you evaluate the trained network on an input word, the output vector will actually be a probability distribution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

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

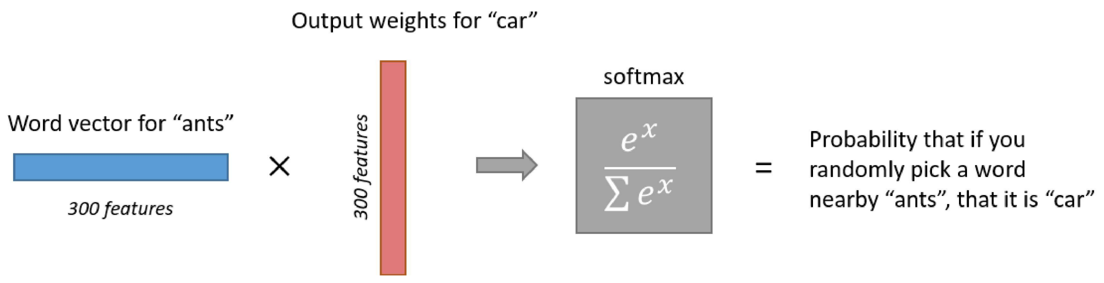
This means that the hidden layer of this model is really just operating as a lookup table. The output of the hidden layer is just the "word vector" for the input word.

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!

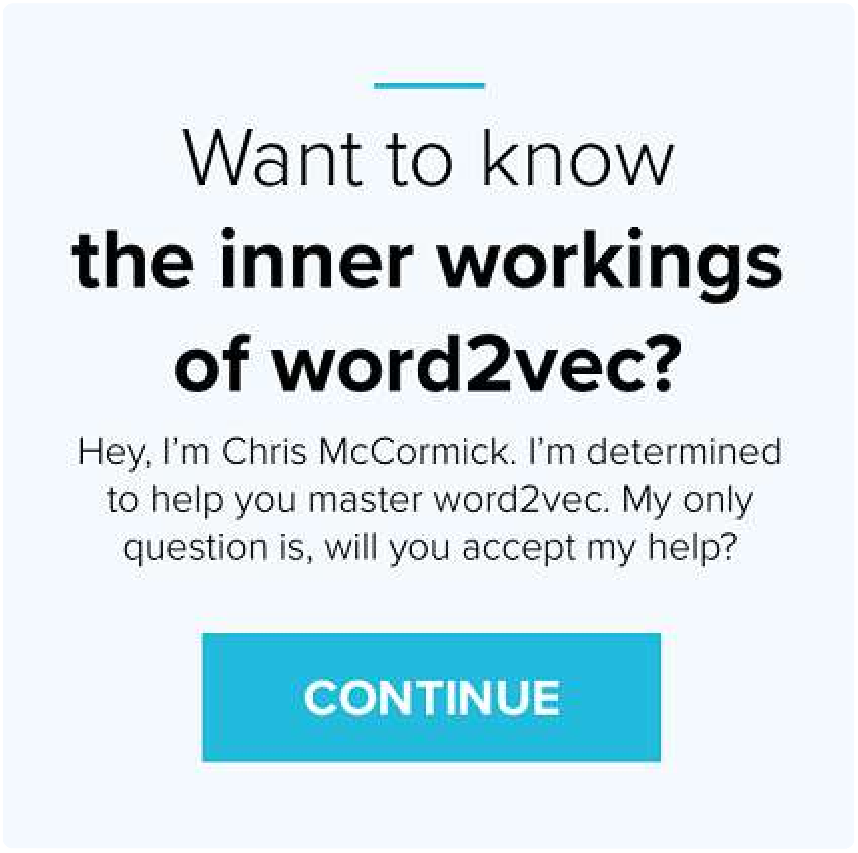
And what does it mean for two words to have similar contexts? I think you could expect that synonyms like "intelligent" and "smart" would have very similar contexts. Or that words that are related, like "engine" and "transmission", would probably have similar contexts as well.

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


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

ALSO ON **MCCORMICKML.COM**

2 years ago • 15 comments
Applying word2vec to Recommenders

4 years ago • 128 comments
Negative Sampling · Chris McCormick

a yea
BE Tur wit

Comments **Community** **Privacy Policy** 1 **Login**

Recommend 204 Tweet Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

raj1514 • 4 years ago • edited
Thanks for this post! It really saved time in going through papers about this...
87 ^ | v • Reply • Share ›

Chris McCormick Mod → raj1514 • 4 years ago
Great! Glad it helped.
3 ^ | v • Reply • Share ›

zul waker • 4 years ago
You are amazing. I tried to understand this from so many sources but you gave the best explanation possible. many thanks.
35 ^ | v • Reply • Share ›

Chris McCormick Mod → zul waker • 4 years ago
Thanks so much, really glad it helped!
7 ^ | v • Reply • Share ›

micsca • 4 years ago

nice article!

26 ^ | ˅ • Reply • Share ›

ningyuwhut • 3 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

25 ^ | ˅ 1 • Reply • Share ›

Supun Abeysinghe → ningyuwhut • 3 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 :)

12 ^ | ˅ • Reply • Share ›

Chris McCormick Mod → Supun Abeysinghe • 2 years ago

I concur!

2 ^ | ˅ • Reply • Share ›

Rajan Saha Raju → Supun Abeysinghe • a month ago

It was actual philosophy...

^ | ˅ • Reply • Share ›

Anne Puzon → ningyuwhut • 2 years 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.

3 ^ | ˅ • Reply • Share ›

Arish Ali • 4 years ago

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

7 ^ | ˅ • Reply • Share ›

Chris McCormick Mod → Arish Ali • 4 years ago

Awesome, thanks!

⌃ | ⌄ • Reply • Share ›

Mike Stopa • 2 years ago

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

6 ⌃ | ⌄ • Reply • Share ›

Chris McCormick Mod → Mike Stopa • 2 years 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.

2 ⌃ | ⌄ • Reply • Share ›

Tank S. • 3 years ago • edited

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?

6 ⌃ | ⌄ • Reply • Share ›

Chris McCormick Mod → Tank S. • 2 years ago

Hey Tank,
The weight matrix is built as part of the neural network training. For the skip-gram model, it's trained on word pairs, but the word pairs are generated from all of the sentences in your training text.
Thanks,
Chris

1 ⌃ | ⌄ • Reply • Share ›

Jonathan Cui • a year ago

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

5 ⌃ | ⌄ • Reply • Share ›

Chris McCormick Mod → Jonathan Cui • a year ago

No bias terms!

On the hidden layer, it's clearly not needed, since this layer just serves as a lookup table for the word vectors. Adding a bias term here would be equivalent to adding one more feature to the word vectors (e.g., making them 301 components instead of 300).

On the output layer, I don't see a bias term in the implementation. Perhaps it's not needed because we're not actually using the output of the model?

The output layer is just a means to an end--it's the hidden layer vectors that we really want.

Thanks,
Chris

3 ^ | v • Reply • Share ›

Albert Wang • 4 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?

7 ^ | v 1 • Reply • Share ›

Leena → Albert Wang • 4 months ago

Hello Albert and Chris, I have the same question:

input sentence -> "I am a software engineer"

X Y
----
a I
a am
a software
a engineer

Now when it trains the first row, it adjust the weights according to cross-entropy loss.
Now for input "a" and output "I", weights are adjusted.

Now for second row, input "a" and output "am", (input is same as first row and output is different) it will again adjust the weight for second row.

Since input is same for first and second row, if we adjust the weights for second row, weight adjustments are overridden for first row.

(This happens because we only have one feature for X and it is repeated.)

What are your thoughts on this?

Leena

Then what happens

^ | ⌄ • Reply • Share ›

Albert Wang → Leena • 4 months ago

It has been years since I left the ML community. Don't remember much. Sorry can't help.

^ | ⌄ • Reply • Share ›

Chris McCormick Mod → Albert Wang • 4 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 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 ›

Andres Suarez → Chris McCormick • 2 years 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!

^ | ⌄ • Reply • Share ›

Albert Wang ➦ Chris McCormick
• 4 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.

^ | ˅ • Reply • Share ›

Anne Puzon ➦ Albert Wang
• 2 years ago

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

1 ^ | ˅ • Reply • Share ›

Kyrill Alyoshin • a year 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.

3 ^ | ˅ • Reply • Share ›

Chris McCormick Mod ➦ Kyrill Alyoshin • a year ago

Thanks Kyrill—we're looking into it!

^ | ˅ • Reply • Share ›

Ashutosh_157 • 2 years ago • edited

I was going through the original paper on word2vec. In section 2.1 first paragraph they have used a term "projection layer". I am unable to understand what are they referring to. I have searched google but could not find a convincing answer.

Any explanation with reference to above article or any other intuitive explanation about "projection layer" will be appreciated.

5 ^ | ˅ 1 • Reply • Share ›

Chris McCormick Mod ➦ Ashutosh_157

• a year ago

Hi Ashutosh,

It's referred to as a "projection layer" because there is no activation function on the hidden layer, only on the output.

First, why don't we need an activation function on the hidden layer? An activation function on the hidden layer is not necessary because the input to the model is a one-hot vector, which only serves to select a word vector from the hidden layer weights.

Here's an illustration of multiplying a one-hot vector with a matrix--note how it does nothing more than selecting a row.

<img src="http://mccormickml.com/asse..."/>

The normal purpose of a non-linear activation function in a neural network is to turn the dot product step, which is only a linear operation, into a non-linear one. This increases the expressive power of the model.

But imagine applying a non-linear activation to the multiplication in the above image--it's pointless!

In short, applying an activation function to the output of the hidden layer is pointless, so we don't, and it's instead referred to as a "projection layer".

Finally, why the name "projection layer"? When we multiply the one-hot vector against the input matrix, we are "projecting" a point from one vector space (the one-hot encoded space) onto another vector space (the word vector space).

There is more discussion on this topic here and here, though I think I've covered all of the insights those offer.

Thanks!
Chris

3 ^ | ˅ • Reply • Share ›

Bob • 4 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-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 • 4 years ago

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

1 ^ | ˅ • Reply • Share ›

Chris McCormick Mod ➦ Bob • 4 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 ›

Gavin Hoang • 2 months ago

Nice。 But one thing I dont understand is that where is the hidden matrix from? How do we generate this matrix? Like those numbers 17, 24, 1, ... how do we get these numbers?

1 ^ | ˅ • Reply • Share ›

Chris Y. • 6 months ago

it's a good article giving a more explanation what the paper doesn't talk about.

But there is a point that is not clear. The 1by300 word vector needs to multiple with the one-hot-encoding of the target word to get a single value to feed to the soft-max with other outputs in the output layer.

But u just said a weight vector... it's not accurate. If it's just a random assigned word vector, where is the information from the target word to related to ? like ant -> insect... insect one-hot-encoding vector must be in the output layer, so the unsupervised training process can maximize the similarity of two words...

1 ^ | ˅ • Reply • Share ›

Chris McCormick Mod → Chris Y. • 6 months ago

Hi Chris, let me see if I can help.

The 1x300 word vector from the hidden layer is fed into the output layer, which has one output neuron for every word in the vocabulary. Each output neuron has its own set of 300 weights. There is an output neuron for "insect", for example.

So the 1x300 hidden layer word vector for "ant" will be multiplied against the 1x300 weights of the output neuron for "insect". This dot-product yields a single value which is then fed in to the SoftMax function. This is repeated for every output neuron.

Did that clarify things?

Thanks,
Chris

1 ^ | ˅ • Reply • Share ›

Chris Y. ➦ Chris McCormick • 6 months ago

Hi Chris,

Thanks a lot for your reply. All what I mentioned is about the output neuron weight which should be a hot-encoding vector of "insect" in the context, but not just a random 1by300 weights as the network is to map "ant" to "insect" by a embedding matrix, so the final output layer neuron should have a target vector. like "insect one-hot-encoding". 1 by 300 weights vector should have 300

Related posts

How to Apply BERT to Arabic and Other Languages 05 Oct 2020
Smart Batching Tutorial - Speed Up BERT Training 29 Jul 2020
GPU Benchmarks for Fine-Tuning BERT 21 Jul 2020