Word2Vec Tutorial Part 2 - Negative Sampling

11 Jan 2017

In part 2 of the word2vec tutorial (here's part 1), I'll cover a few additional modifications to the basic skip-gram model which are important for actually making it feasible to train.

When you read the tutorial on the skip-gram model for Word2Vec, you may have noticed something–it's a huge neural network!

In the example I gave, we had word vectors with 300 components, and a vocabulary of 10,000 words. Recall that the neural network had two weight matrices–a hidden layer and output layer. Both of these layers would have a weight matrix with 300 x 10,000 = 3 million weights each!

Running gradient descent on a neural network that large is going to be slow. And to make matters worse, you need a huge amount of training data in order to tune that many weights and avoid over-fitting. millions of weights times billions of training samples means that training this model is going to be a beast.

The authors of Word2Vec addressed these issues in their second paper with the following two innovations:

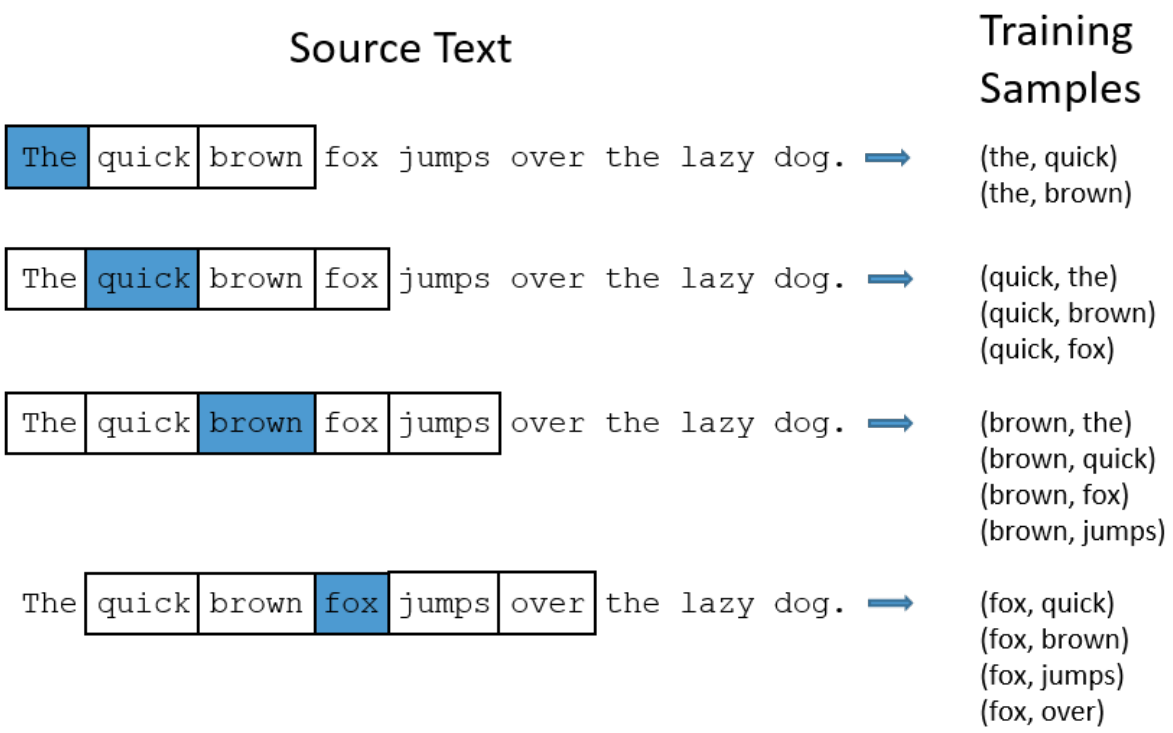
1. Subsampling frequent words to decrease the number of training examples.

2. Modifying the optimization objective with a technique they called "Negative Sampling", which causes each training sample to update only a small percentage of the model's weights.

It's worth noting that subsampling frequent words and applying Negative Sampling not only reduced the compute burden of the training process, but also improved the quality of their resulting word vectors as well.

Subsampling Frequent Words

In part 1 of this tutorial, I showed how training samples were created from the source text, but I'll repeat it here. The below example shows some of the training samples (word pairs) we would take from the sentence "The quick brown fox jumps over the lazy dog." I've used a small window size of 2 just for the example. The word highlighted in blue is the input word.



There are two "problems" with common words like "the":

1. When looking at word pairs, ("fox", "the") doesn't tell us much

about the meaning of “fox”. “the” appears in the context of pretty much every word.

2. We will have many more samples of (“the”, …) than we need to learn a good vector for “the”.

Word2Vec implements a “subsampling” scheme to address this. For each word we encounter in our training text, there is a chance that we will effectively delete it from the text. The probability that we cut the word is related to the word’s frequency.

If we have a window size of 10, and we remove a specific instance of “the” from our text:

1. As we train on the remaining words, “the” will not appear in any of their context windows.
2. We’ll have 10 fewer training samples where “the” is the input word.

Note how these two effects help address the two problems stated above.

Sampling rate

The word2vec C code implements an equation for calculating a probability with which to keep a given word in the vocabulary.

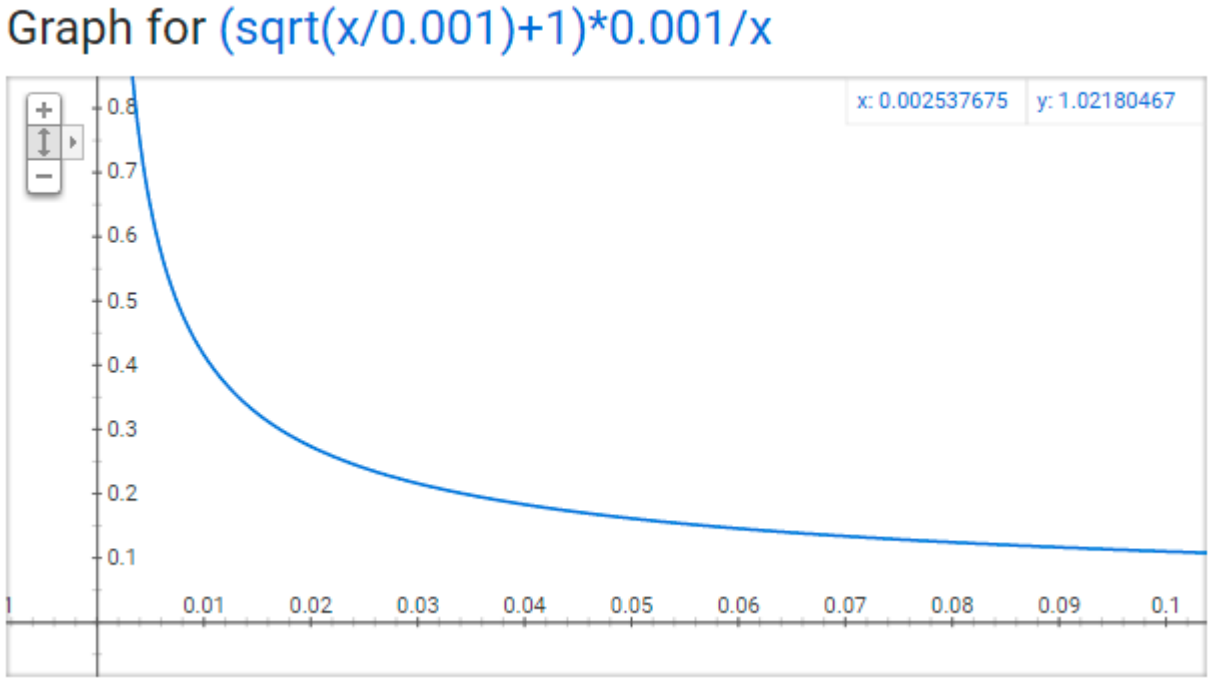
w_i is the word, $z(w_i)$ is the fraction of the total words in the corpus that are that word. For example, if the word “peanut” occurs 1,000 times in a 1 billion word corpus, then z(‘peanut’) = 1E-6.

There is also a parameter in the code named ‘sample’ which controls how much subsampling occurs, and the default value is 0.001. Smaller values of ‘sample’ mean words are less likely to be kept.

$P(w_i)$ is the probability of *keeping* the word:

$$P(w_i) = (\sqrt{\frac{z(w_i)}{0.001}} + 1) \cdot \frac{0.001}{z(w_i)}$$

You can plot this quickly in Google to see the shape.



No single word should be a very large percentage of the corpus, so we want to look at pretty small values on the x-axis.

Here are some interesting points in this function (again this is using the default sample value of 0.001).

- $P(w_i) = 1.0$ (100% chance of being kept) when $z(w_i) <= 0.0026$.
  - This means that only words which represent more than 0.26% of the total words will be subsampled.

- $P(w_i) = 0.5$ (50% chance of being kept) when $z(w_i) = 0.00746$.
- $P(w_i) = 0.033$ (3.3% chance of being kept) when $z(w_i) = 1.0$.
  - That is, if the corpus consisted entirely of word w_i, which of course is ridiculous.

You may notice that the paper defines this function a little differently than what's implemented in the C code, but I figure the C implementation is the more authoritative version.

Negative Sampling

Training a neural network means taking a training example and adjusting all of the neuron weights slightly so that it predicts that training sample more accurately. In other words, each training sample will tweak *all* of the weights in the neural network.

As we discussed above, the size of our word vocabulary means that our skip-gram neural network has a tremendous number of weights, all of which would be updated slightly by every one of our billions of training samples!

Negative sampling addresses this by having each training sample only modify a small percentage of the weights, rather than all of them. Here's how it works.

When training the network on the word pair ("fox", "quick"), recall that the "label" or "correct output" of the network is a one-hot vector. That is, for the output neuron corresponding to "quick" to output a 1, and for *all* of the other thousands of output neurons to output a 0.

With negative sampling, we are instead going to randomly select just a small number of "negative" words (let's say 5) to update the weights for. (In this context, a "negative" word is one for which we want the network to output a 0 for). We will also still update the weights for our "positive" word (which is the word "quick" in our current example).

The paper says that selecting 5-20 words works well for smaller datasets, and you can get away with only 2-5 words for large

datasets.

Recall that the output layer of our model has a weight matrix that's 300 x 10,000. So we will just be updating the weights for our positive word ("quick"), plus the weights for 5 other words that we want to output 0. That's a total of 6 output neurons, and 1,800 weight values total. That's only 0.06% of the 3M weights in the output layer!

In the hidden layer, only the weights for the input word are updated (this is true whether you're using Negative Sampling or not).

Selecting Negative Samples

The "negative samples" (that is, the 5 output words that we'll train to output 0) are selected using a "unigram distribution", where more frequent words are more likely to be selected as negative samples.

For instance, suppose you had your entire training corpus as a list of words, and you chose your 5 negative samples by picking randomly from the list. In this case, the probability for picking the word "couch" would be equal to the number of times "couch" appears in the corpus, divided the total number of word occus in the corpus. This is expressed by the following equation:

$$P(w_i) = \frac{f(w_i)}{\sum_{j=0}^{n}\left(f(w_j)\right)}$$

The authors state in their paper that they tried a number of variations on this equation, and the one which performed best was to raise the word counts to the 3/4 power:

$$P(w_i) = \frac{f(w_i)^{3/4}}{\sum_{j=0}^{n}\left(f(w_j)^{3/4}\right)}$$

If you play with some sample values, you'll find that, compared to the

simpler equation, this one has the tendency to increase the probability for less frequent words and decrease the probability for more frequent words.

The way this selection is implemented in the C code is interesting. They have a large array with 100M elements (which they refer to as the unigram table). They fill this table with the index of each word in the vocabulary multiple times, and the number of times a word's index appears in the table is given by $P(w_i)$ * table_size. Then, to actually select a negative sample, you just generate a random integer between 0 and 100M, and use the word at that index in the table. Since the higher probability words occur more times in the table, you're more likely to pick those.

Word Pairs and "Phrases"

The second word2vec paper also includes one more innovation worth discussing. The authors pointed out that a word pair like "Boston Globe" (a newspaper) has a much different meaning than the individual words "Boston" and "Globe". So it makes sense to treat "Boston Globe", wherever it occurs in the text, as a single word with its own word vector representation.

You can see the results in their published model, which was trained on 100 billion words from a Google News dataset. The addition of phrases to the model swelled the vocabulary size to 3 million words!

If you're interested in their resulting vocabulary, I poked around it a bit and published a post on it here. You can also just browse their vocabulary here.

Phrase detection is covered in the "Learning Phrases" section of their paper. They shared their implementation in word2phrase.c–I've shared a commented (but otherwise unaltered) copy of this code here.

I don't think their phrase detection approach is a key contribution of their paper, but I'll share a little about it anyway since it's pretty straightforward.

Each pass of their tool only looks at combinations of 2 words, but you can run it multiple times to get longer phrases. So, the first pass will pick up the phrase "New_York", and then running it again will pick up "New_York_City" as a combination of "New_York" and "City".

The tool counts the number of times each combination of two words appears in the training text, and then these counts are used in an equation to determine which word combinations to turn into phrases. The equation is designed to make phrases out of words which occur together often relative to the number of individual occurrences. It also favors phrases made of infrequent words in order to avoid making phrases out of common words like "and the" or "this is".

You can see more details about their equation in my code comments here.

> One thought I had for an alternate phrase recognition strategy would be to use the titles of all Wikipedia articles as your vocabulary.

Other Resources

If you're familiar with C, I've published an extensively commented (but otherwise unaltered) version of the original word2vec C code here.

Also, 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.

eBook & Example Code

I think word2vec is a fascinating (and powerful!) algorithm–great work on making it this far in understanding it!

Maybe you still have some questions, though…

- Are you looking for a deeper explanation of how the model weights are updated?
- Would you like to know more about the technical and practical differences between the Skip-gram and Continuous Bag of Words (CBOW) versions of word2vec?
- Did you know that Mikolov, the main author of word2vec, has published further work on word2vec in the form of the fastText library from Facebook?
- Want to see all of the core word2vec components implemented from scratch in Python?

You'll find all of the above content in my eBook The Inner Workings of word2vec. Give it a look, I think you'll find it really valuable!

Cite

McCormick, C. (2017, January 11). *Word2Vec Tutorial Part 2 - Negative Sampling*. Retrieved from http://www.mccormickml.com

Yueting Liu • 5 years ago

I have got a virtual map in my head about word2vec within a couple hours thanks to your posts. The concept doesn't seem daunting anymore. Your posts are so enlightening and easily understandable. Thank you so much for the wonderful work!!!

33 • Reply •

Chris McCormick Mod ➚ Yueting Liu • 5 years ago

Awesome! Great to hear that it was so helpful--I enjoy writing these tutorials, and it's very rewarding to hear when they make a difference for people!

10 • Reply •

1mike12 ➚ Yueting Liu • 4 years ago

I agree, shit is lit up cuz

1 • Reply •

Mahnaz K • 4 years ago

Hi Chris, I hope that you'll still read and respond to comments on this post sometimes. Great post, also appreciate clarification made in comments. I've a question for you. When you're explaining the 3/4 power for weights, you're referring to range(0,1). But the X axis is the word frequency which wont fall in that range. Shouldn't we be focused on the rest of plot and note that raising frequency to the power of 3/4 will change the distribution the way that words with their frequencies being in a (wide) range would be treated the same way? - almost like bucketing!

33 2 • Reply •

Chris McCormick Mod ➚ Mahnaz K • 3 years ago

Hi Mahnaz, thank you for your comment! That portion of the post was in error, and I've updated it.

In order to draw insights from the negative sampling equation, it helps to apply it to some real word frequency data. You can generate a plot where the x-axis is the word's ranking in the vocabulary, and the y-axis is the probability of selecting it. The gist of it is that it **decreases** the probability of choosing **common** words and **increases** the probability of choosing **rarer** words.

I go into depth on this in Chapter 3 of the eBook, and in the accompanying example code, both of which are available to purchase here.

Thanks again!
Chris
3 • Reply •

binspiredmamakrissy Mahnaz K • 3 years ago
mahnaz_k more
• Reply •

Smith Jason Mahnaz K • 3 years ago
In my opinion, I think you can limit the range of the f(w_i) to (0,1) by dividing N_3/4 on both denominator and numerator. N is the sum of the word frequency of the whole words in the training vocabulary. Instead of taking f(w_i) as the word frequency, you can see it as the the fraction of the total words in the corpus that are that word.
• Reply •

Laurence Obi • 4 years ago
Hi Chris,
Awesome post. Very insightful. However, I do have a question. I noticed that in the bid to reduce the amount of weights we'll have to update, frequently occurring words were paired and viewed as one word. Intuitively, that looks to me like we've just added an extra word (New York) while other versions of the word New that have not occurred with the word York would be treated as stand-alone. Am I entirely wrong to assume that the ultimate size of our one-hot encoded vector would grow in this regard?
Thanks.
10 • 1 • Reply •

Mahnaz K Laurence Obi • 4 years ago
"Common word pair" is an improvement to the original model in a way to increase the performance of the model. As it was mentioned in the post the context for "New" as a single word is different than the context for "New-York". Unlike other improvements presented in second paper, this one brings a cost along. Still it's worth it.
3 • Reply •

Chris McCormick **Mod** Laurence Obi • 3 years ago
Hi Laurence, thanks for your comment! Mahnaz is correct--adding word pairs and phrases to the vocabulary makes the compute burden higher, not less.

I see how the post was misleading around this, though. I've moved the **Word Pairs and "Phrases"** section to the end, so that it's not misconstrued as a performance enhancement (like the other two topics).

Thanks again!
Chris
• Reply •

amine abnine • 3 years ago

this is probably one of the best articles i've read so far in the matter of word2vec and how it was optimized. Thank you very much Chris!

6 | • Reply •

Chris McCormick **Mod** → amine ahnine • 3 years ago

Awesome, thanks for the encouragement!!

| • Reply •

Octavian Bordeanu • 3 years ago

Hi. Great tutorial and thank you. I was wondering what is the resource paper for the following:

"They have a large array with 100M elements (which they refer to as the unigram table). They fill this table with the index of each word in the vocabulary multiple times, and the number of times a word's index appears in the table is given by P(wi) * table_size."

I am interested to know why 100 million was chosen as the table size.

4 | • Reply •

Chris McCormick **Mod** → Octavian Bordeanu • 3 years ago

Hi Octavian, thanks for the comment!

The size of the unigram table determines the precision of its behavior. Whatever the probability is for a word, you need to round it to an integer number of table rows, which means some loss of precision.

The size of the vocabulary also matters--their Google News model had *3 million words*, so I think 100M is scaled for that vocabulary size.

These numbers come from the author's own C implementation, which I've commented . Search for the "InitUnigramTable" function.

Thanks,
Chris

1 | • Reply •

fangchao liu • 4 years ago

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

2 | • Reply •

lovekesh thakur → fangchao liu • 3 years ago

It's probabilistic model. So, if we have a large vocabulary, the probability of

occurring this will be small.

1 ^ | v • Reply •

Jane • 5 years ago

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

2 ^ | v • Reply •

Chris McCormick **Mod** ➚ Jane • 5 years ago

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

^ | v • Reply •

Sebastiaan Van Baars • 2 years ago

this made a big difference for me in terms of understanding the optimisation process. Thanks!

1 ^ | v • Reply •

Chris McCormick **Mod** ➚ Sebastiaan Van Baars • 2 years ago

Good to hear, thanks Sebastian!

^ | v • Reply •

Raki Lachraf • 3 years ago

exactly what I needed to know about Negative Sampling thank you so much may God bless you !

1 ^ | v • Reply •

Joey Bose • 4 years ago

So the subsampling P(w_i) is not really a probability as its not bounded between 0-1. Case in point try it for 1e-6 and you get a 1000 something, threw me for quite a loop when i was coding this.

1 ^ | v • Reply •

Chris McCormick **Mod** ➚ Joey Bose • 4 years ago

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

^ | v • Reply •

Ben Bowles • 5 years ago

Thanks for the great tutorial.

About this comment "Recall that the output layer of our model has a weight matrix that's 300 x 10,000. So we will just be updating the weights for our positive word ("quick"), plus the weights for 5 other words that we want to output 0. That's a total of 6 output neurons, and 1,800 weight values total. That's only 0.06% of the 3M weights in the output layer!"

Should this actually be 3600 weights total for each training example, given that we have an embedding matrix and an matrix of weights, and BOTH involve updating 1800 weights (300 X 6 neurons)? (Both of which should be whatever dimension you are using for your embeddings multiplied by vocab size)?

1 • Reply •

Chris McCormick Mod ➚ Ben Bowles • 5 years ago

Hi Ben, thanks for the comment.

In my comment I'm talking specifically about the output layer. If you include the hidden layer, then yes, there are more weights updated. The number of weights updated in the hidden layer is only 300, though, not 1800, because there is only a single input word.

So the total for the whole network is 2,100. 300 weights in the hidden layer for the input word, plus 6 x 300 weights in the output layer for the positive word and five negative samples.

And yes, you would replace "300" with whatever dimension you are using for your word embeddings. The vocabulary size does *not* factor into this, though--you're just working with one input word and 6 output words, so the size of your vocabulary doesn't impact this.

Hope that helps! Thanks!

1 • 1 • Reply •

Ben Bowles ➚ Chris McCormick • 5 years ago

This is super helpful, I appreciate this. My intuition (however naive it may be) was that the embeddings in the hidden layer for the negative sample words should also be updated as they are relevant to the loss function. Why is this not the case? I suppose I may have to drill down into the equation for backprop to find out. I suppose it has to do with the fact that when the one-hot vector is propagated forward in the network, it amounts to selecting only the embedding that corresponds to the target word.

• Reply •

Chris McCormick Mod ➚ Ben Bowles • 5 years ago • edited

That's exactly right--the derivative of the model with respect to the weights of any other word besides our input word is going to be zero.

Hit me up on LinkedIn!

1 • Reply •

Leland Milton Drake ➚ Chris McCormick • 4 years ago

Hey Chris,

When you say that only 300 weights in the hidden layer are updated, are you assuming that the training is done with a minibatch of size 1?

I think if the minibatch is greater than 1, then the number of weights that will be updated in the hidden layer is 300 x number

of unique input words in that minibatch.

Please correct me if I am wrong.

And thank you so much for writing this post. It makes reading the academic papers so much easier!

53 ^ | v • Reply •

Chris McCormick **Mod** ➚ Leland Milton Drake • 4 years ago • edited

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

1 ^ | v • Reply •

Jack Gao • 5 months ago

Thank you for your detailed explanation! This is the best personal knowledge-sharing website I have ever seen! :D

^ | v • Reply •

CyberDreamer • 2 years ago

Thanks for tutorial!

I've thought for a long time that negative sampling is some tricky way to calculate the loss function. But it turns out that is a mechanism which limits the movement of the gradient through a large number of weights (neurons). This means multiplying a small matrix. That is, the mechanism works inside the network, and not somewhere outside. I'm right?

But why this gradient not important? Why the same mechanism not applied, say in computer vision?

^ | v • Reply •

yubinml • 2 years ago

Dost The Inner Workings of word2vec - Pro Edition has python code step by step?

^ | v • Reply •

Darbhamulla Eswara Phani P • 2 years ago

Great explanation !

^ | v • Reply •

Bhairaja Gautam • 2 years ago

" In the hidden layer, only the weights for the input word are updated (this is true whether you’re using Negative Sampling or not) ". When we backpropagate, the weights of other words i.e. weight matrix corresponding to other words are also updated, isn't it?

^ | v • Reply •

snovaisg • 2 years ago • edited

In regards to the Subsampling, I was still trying to understand the intuition behind the formula for keeping/ removing each word so i decided to play with some plots (link on

the bottom of post):

First, it makes sense that we don't want to use common (aka: high frequency) words such as "that, the, etc..", so the formula for the Probability of accepting a word should be inversely proportional to the frequency of the word, which leads us to the **naive formula: P(wi) = 1/f(wi)**, where f(wi) is the word i's frequency. This is a decreasing function as f(wi) goes from 0 to 1. (the wide red line in the plot)

Then we realise that the frequency domain is between [0,1] which means the naive formula P(wi) is always greater than 1. no matter the frequency. But we want to have words whose P(wi) is less than 1, and ideally, very small when those words are very frequent. Therefore we can add an extra term, the **sample** parameter, **t** which, if you notice in the figure below, will make the function P(wi) drop sooner. actually, when f(wi) reaches **s**, P(wi) = 1, and then it drops further. Our formula for the probability of keeping the word i is now: **P(wi) = t/f(wi)**. (blue line)

The authors then decide to add the square root which has a smoothing effect (purple

see more

• Reply •

Dum • 2 years ago

Hi,
great explanation.

I need python implementation of skip gram negative sampling

• Reply •

Nikhil Tirumala ➚ Dum • 2 years ago

I havent checked this implementation yet, but this might help:
https://www.kaggle.com/ashu...

• Reply •

Harry Richard • 2 years ago

Thank You so much!!!!Continue the good work!!

• Reply •

perrohunter • 3 years ago

Great article. you have a typo on the second paragraph of `Selecting Negative Samples`, it says `occus`.

• Reply •

Louis • 3 years ago

This article is pure gold. Thanks Chris

• Reply •

Mingxuan Cao • 3 years ago

hi, thanks for your sharing. You mentioned "In the hidden layer, only the weights for the input word are updated (this is true whether you're using Negative Sampling or not)". I

think you are here referring to the input weights matrix, right? How about for the 300x6 output weights matrix when using Negative Sampling? How do you get the weights for this 300x6 matrix? Is it copied from the input weights matrix for these 6 words? Because if only the weights for the input word are updated, then this 300x6 matrix will never be updated. Just wonder where do we get it.

• Reply •

Nikhil Tirumala ➚ Mingxuan Cao • 2 years ago

I might be wrong but:
Q. I think you are here referring to the input weights matrix, right?
A. Yes
Q. Is it copied from the input weights matrix for these 6 words?
A. No, the output weights are different. These are discarded after the training is over and only the input weights are kept.
Inputs shape: 1x 10000 # for 10000 words
W1: Input Weights shape: 10000 x 300 # 300 is the embedding size.
W2: Output Weights shape: 300 x 10000 # without negative sampling.
Both W1 and W2 get updated. We discard W2 after training and keep only W1 which are nothing but the embeddings for the words.
In case of Negative sampling, out of 10000 only 6 are selected for loss and backpropagation.
Hope this helps.

• Reply •

Dhruv Bhargava • 3 years ago

In subsampling, do we remove all the instances of the words which have selection probability less than 1 or do we remove the corresponding amount of words (for ex. if the selection probability of a word is .5 then do we remove 50% of it's instances??) also if we do remove all the instances then how are the representations for those words learnt ??

• Reply •

Nikhil Tirumala ➚ Dhruv Bhargava • 2 years ago

I am still learning this, so i might be wrong, but:
If the selection prob is 0.5, then we discard 50% of the samples and keep the rest. I don't think we will remove all the instances.
Also, despite this, there might still be words that are not in dataset or we just ignore them since the frequency might be too low. In all such cases, it will be considered as OOV - out of vocabulary word. Probably word with index '0' in the dataset will be OOV (or UNK)

• Reply •

Clive777 • 3 years ago

I have a silly qts if my input is quick
Then my output should be
The ,fox , brown rite ?

• Reply •

Nikhil Tirumala ➚ Clive777 • 2 years ago

The quick brown fox jumps....
If your input is quick, then output will be 'The' and 'brown'.

^ | v • Reply •

mlwhiz • 3 years ago

This is the most simple explanation of the topic that I found. Thanks for taking out the time to write this,

^ | v • Reply •

trault14 • 3 years ago

Thanks again Chris for this awesome explanation !
I have a question. Why is the length of the unigram table chosen to be 100M elements in the C-code implementation ? Does 100M have a special meaning relative the vocabulary size or the size of the hidden layer ?

^ | v • Reply •

Ali KebariGhotbi • 3 years ago

Chris, Awesome explanation. You have saved us thousands of man hours putting these together. I have a little point which might be worth adding. The objective function

Related posts

Combining Categorical and Numerical Features with Text in BERT 29 Jun 2021

How To Build Your Own Question Answering System 27 May 2021

2020 NLP and NeurIPS Highlights 23 Mar 2021