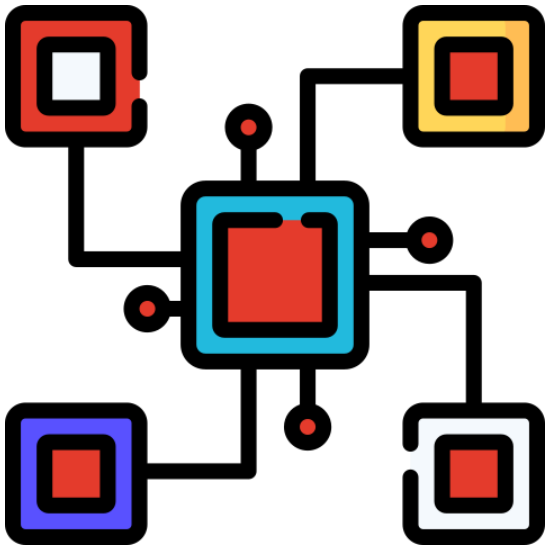
The Inner Workings

- of -

word2vec



By Chris McCormick

It is my earnest desire that the information in this book be as correct as possible; however, I cannot make any guarantees. This is an evolving book about an evolving technology in an evolving field--there are going to be mistakes! So here's my disclaimer: The author does not assume and hereby disclaims any liability to any party for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from negligence, accident, or any other cause.

Edition: v2.0.0

Contents

Introduction

Welcome to my word2vec eBook! Whether you are a student learning important machine learning concepts, a researcher exploring new techniques and ideas, or an engineer with a vision to build a new product or feature, my hope is that the content in this guide will help you gain a deeper understanding of the algorithm, and equip you to realize your own goals faster and with better results.

Here is an overview of the content you'll find in this book.

Chapter 1 - Word Vectors & Their Applications

- This chapter will answer the questions, "what is a word vector?" and "how are they useful?" I'll explain how word vectors can be used to measure how similar two words are in meaning, and the value this has across a number of applications. You may skip this section if you are already familiar with the motivations and uses for word vectors.

Chapter 2 - Skip-gram Model Architecture

- After learning why word vectors are valuable, Chapter 2 will address how (both conceptually and in implementation) the word2vec approach is able to learn and encode the meaning of a word.

Chapter 3 - Sampling Techniques

- The architecture described in chapter 2 is good in concept but prohibitively expensive in practice. Negative Sampling is a slight modification to the training process which is both dramatically faster and produces higher quality results.

Chapter 4 - Model Variations

- For completeness, chapter 4 describes the Continuous Bag-of-Words (CBOW) architecture (an alternative to the skip-gram architecture which was also presented in the original word2vec paper), and Hierarchical Softmax (an alternative to Negative Sampling).

Chapter 5 - Backpropagation Training

- Chapter 5 walks through the training process in more detail, showing the specifics of the weight updates performed with each training sample.

Chapter 6 - Subword Vectors & fastText

- Chapter 6 explores a new variant of the word2vec model which learns vectors for *parts of words*, in addition to whole words, and then uses these added vectors to learn quality vectors with less training data. This word2vec variant was published along with Facebook's "fastText" library for word model training.

Appendix A - FAQ

- The FAQ section addresses some common questions (and some common sources of confusion!) around word2vec.

Appendix B - Resources

- This section points to further helpful resources:
    1. The original papers and implementation.
    2. Articles which explain the mathematical formulation.
    3. A brief survey of some popular implementations of word2vec.

Example Code

- There is Python example code to go along with most chapters of this eBook (if you have not already done so, you can purchase the code here). At the end of each chapter in this book, you'll find a summary of the corresponding example code.

Feedback

- I want your feedback! If you have questions, if you spot any mistakes, if you have suggestions for additional content, or if you have any other feedback, please drop a note in the comments here!

1. Word Vectors & Their Applications

1.1. What's a Word Vector?

If you are already familiar with the concept of feature vectors, and the motivations for creating word vectors, then feel free to skip on to the next chapter.

Let's start by thinking abstractly about what word vectors are, and then we'll get into the details.

Word vectors, also called **word embeddings**, represent words in a way that: (1) encodes their meaning and (2) allows us to calculate a similarity score for any pair of words. The similarity score is simply a fractional value between -1.0 and 1.0, with higher values corresponding to higher similarity.

Along with their academic papers, the authors released a pre-trained model that was trained on a large Google News dataset. This Google News model contains vectors for 3 million words (including multi-word names or concepts like "Elon_Musk" or "computer_science"). If I take their vector representing the word "couch", and their vector representing the word "book", I can do some math (which I'll get to in a minute) to compare their vectors. This gives a score of `0.12`--not very similar. If I do the same thing with "couch" and "sofa", I get a score of `0.83`, reflecting the fact that these words are synonyms.

Being able to compare two words in this way also means that you can compare a word to an entire *vocabulary* of words, and identify the most similar words in that vocabulary. For example, the three most similar words to "couch" in Google's model are "sofa", "recliner", and "couches".

If word2vec was nothing more than a thesaurus for identifying synonyms, that may not be that useful. But it also recognizes related

topics; for example, ”Abraham_Lincoln” and “Gettysburg_Address” have a similarity score of `0.60`.

Even better, a word model can be trained on a specific collection of text, and learn the meanings of words specific to that context. For instance, Enron notoriously used Star-Wars-inspired codenames to name some of their “off the books” projects. A word model trained on the Enron email corpus can show you that “jedi” and “off-the-books” are related terms! (source)

Word models can be helpful in a variety of applications dealing with text. For instance, if you are searching a big collection of documents for the term “home_mortgage”, it would be helpful if your search tool could recommend similar concepts to search for, like “mortgage_loan”, “underwriter”, “down_payment”, …

Or what if you could train a word model that is able to recognize that the words “adios” and “goodbye” have the same meaning? Word models have been created with this property to help in automatically translating text (“machine translation”).

1.2. Feature Vectors & Similarity Scores

Word vectors represent words in a way that encodes their meaning. A vector is simply an array of fractions. Here’s a vector taken from the Google News model. It’s the vector for the word “fast”. It consists of 300 floating point values (I’ve only displayed a handful of them):

```
<0.0575, -0.0049, 0.0474, ..., -0.0439>
```

The individual values don't really correspond to anything that you can interpret. Instead, you should think of a word vector as a point in high-dimensional space.

We're familiar with 2D coordinates and 3D coordinates. We can visualize these, and we intuitively understand the concept of distance between points--pairs of points can be near together or far apart.

But a word vector is a 300-dimensional coordinate... We obviously can't visualize 300-dimensional space coordinates, but the notion of distance between points is still true.

The straight line distance between two points is called the "Euclidean" or "L2" distance, and it can be extended to any number of dimensions. Here's the formula for the distance between two vectors *a* and *b* with an arbitrary number of dimensions (*n* dimensions):

$$dist_{L2}(a,\ b) = \sqrt{\sum_{i=0}^{n}(a_i - b_i)^2}$$

Here's the important insight--word2vec learns a vector for each word in a vocabulary such that words with similar meanings are *close together* and words with different meanings are *farther apart.* That is, the Euclidean distance between a pair of word vectors becomes a measure of how dissimilar they are.

But wait, didn't we talk earlier about word vector comparisons in terms of a *similarity* score, from -1.0 to 1.0? The Euclidean distance is what we intuitively understand as distance, and it's a workable **distance metric** for comparing word vectors, but in practice another metric called the **Cosine similarity** gives better results.

The Cosine similarity between two vectors a and b is found by calculating their dot product, and dividing this by their magnitudes. The cosine similarity is always a value between -1.0 and 1.0.

$$sim_{cos}(a, b) = \frac{a \cdot b}{\|a\|\|b\|}$$

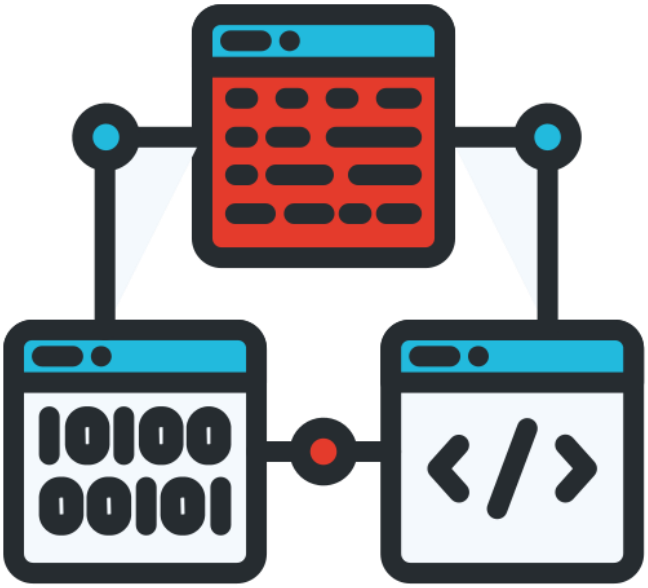
If you'd like to learn more about the vocabulary in the Google News model, and how they identified multi-word names and topics, this topic is covered in more detail in the FAQ.

1.3 Example Code Summary

In this chapter's notebook we'll play around with a pre-trained word model to look at its vocabulary and to try out some of the basic operations commonly performed on word vectors.

We'll start by using the Python package gensim which implements all of the basic features we need like loading the model, accessing its vocabulary, and performing similarity lookups. Immediately after, though, we'll go "under the covers" and perform the same operations manually so you can see what's really going on.

2. Skip-gram Model Architecture



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.

2.1. 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.

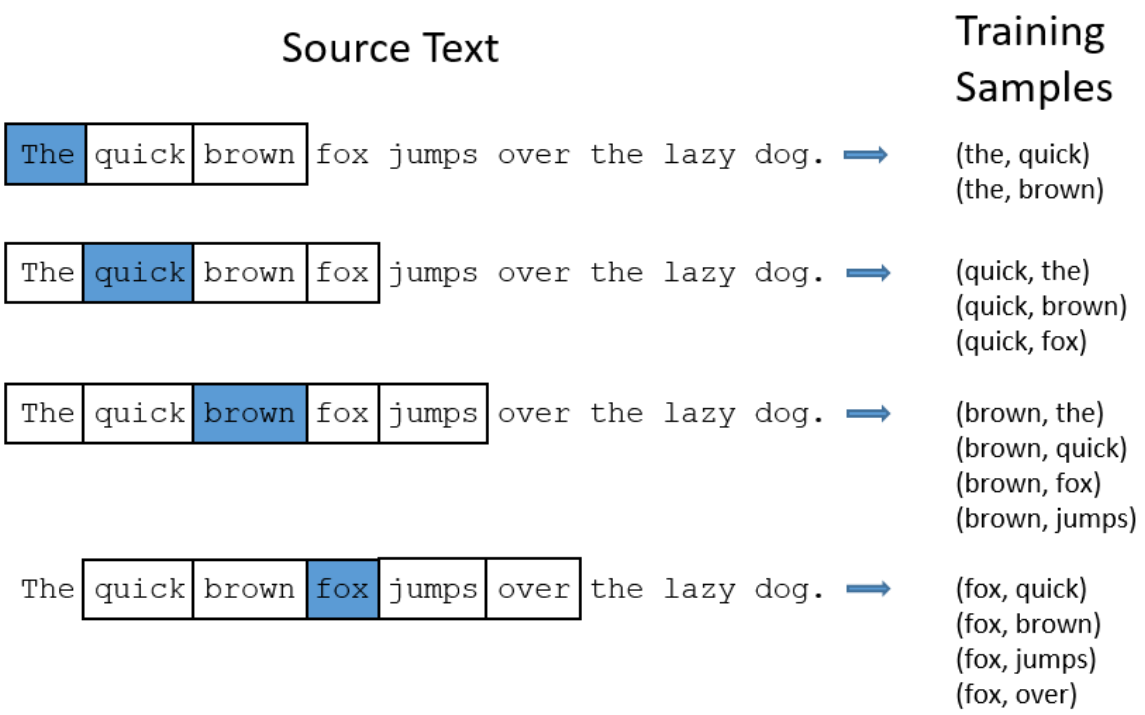
We're going to train the neural network to do the following. Given a specific word in the middle of a sentence (the input word), look at the words nearby and pick one at random. The network is going to tell us

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".

During training, word2vec actually picks a *random window size* between 1 and `window_size`. This has the (indirect) effect of giving different weight to the context words based on their distance from the center word. We'll revisit this in the next chapter.

2.2. Model Details

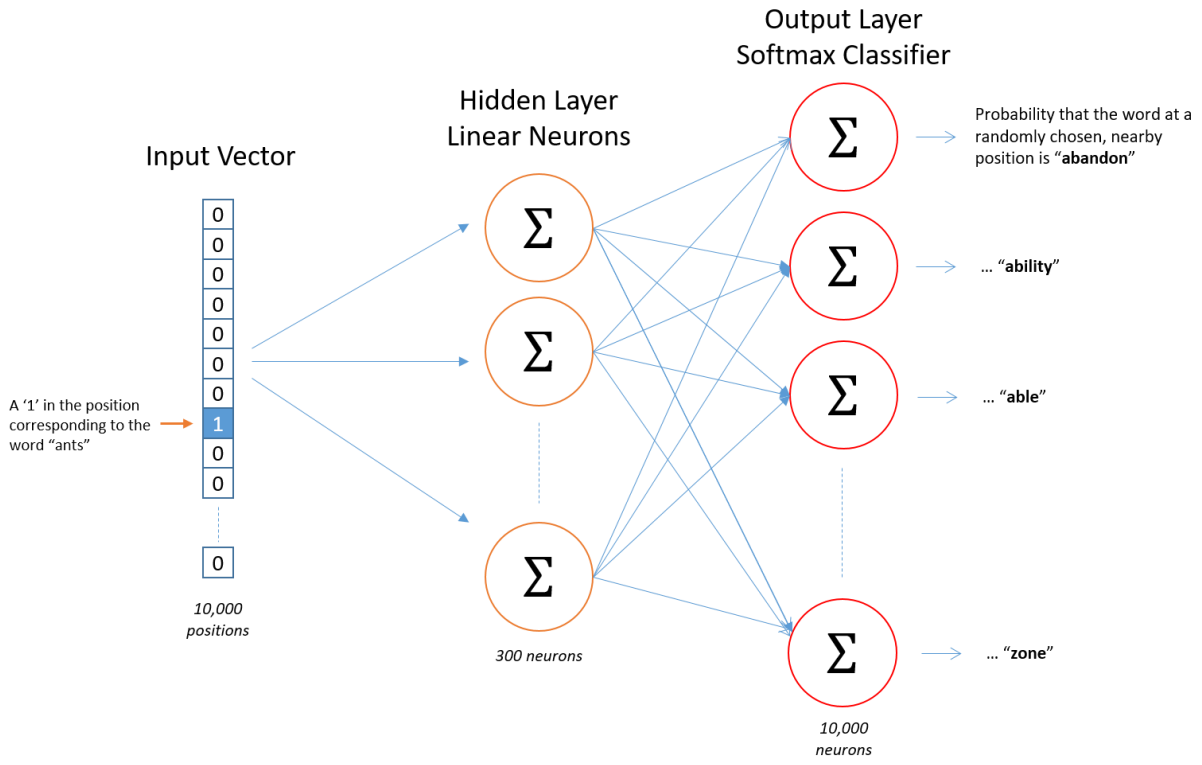
So how is this all represented?

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.

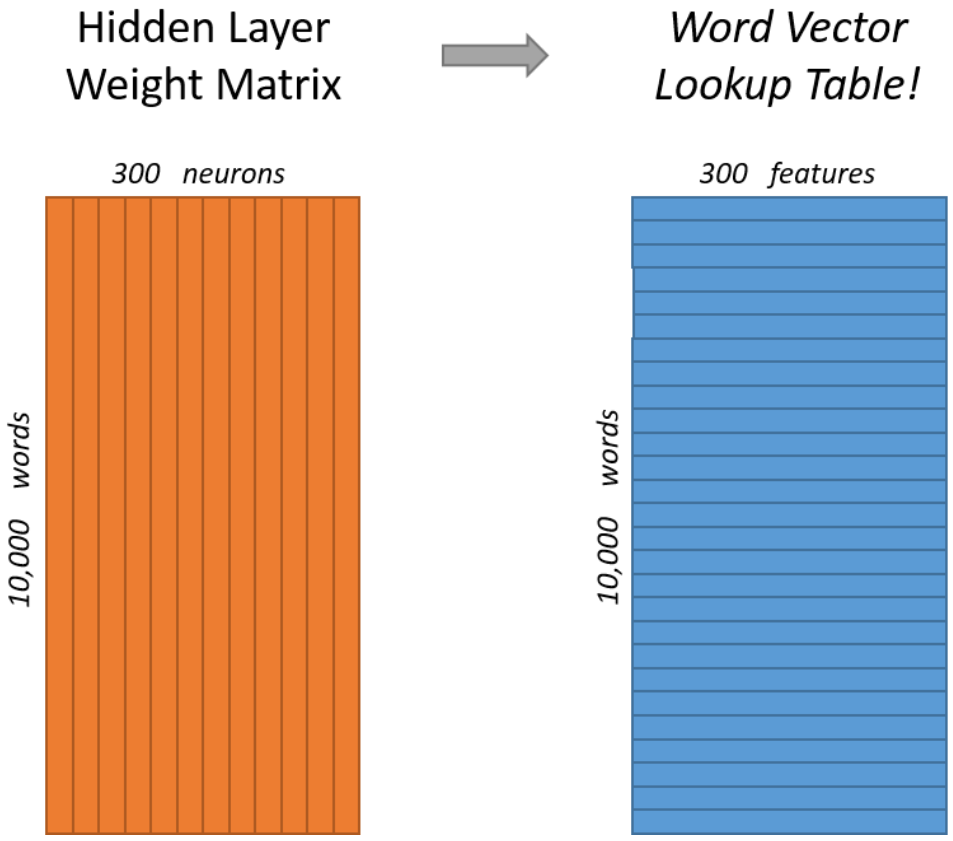
When training this network on word pairs, the input is a one-hot vector representing the input word and the training output is also a one-hot vector representing the output word. But when you evaluate the trained network on an input word, the output vector will actually be a probability distribution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

2.3. 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.

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

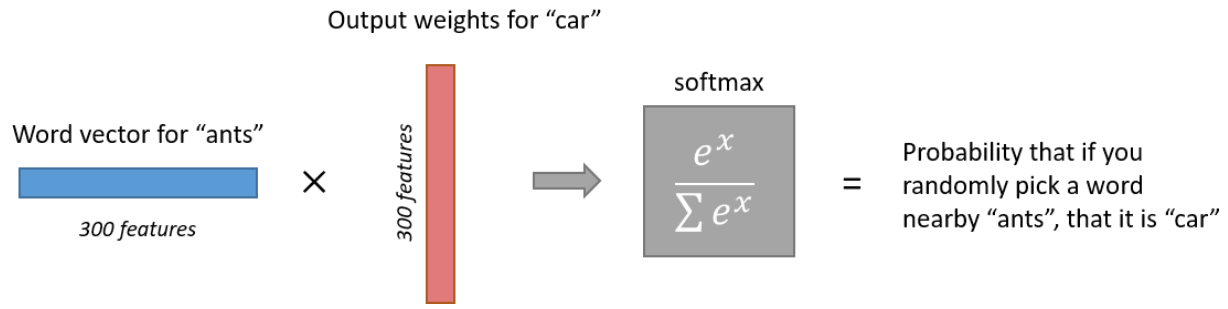
This means that the hidden layer of this model is really just operating as a lookup table. The output of the hidden layer is just the "word vector" for the input word.

2.4. 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".



2.5. 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.

"Fake" Tasks

We started this chapter by explaining that word2vec involves training a neural network on a task that we won't ultimately use it for. This is a powerful technique that is used broadly in machine learning: pick a bogus task for which *training data is plentiful*, train a neural network on it, and then strip off the last layer (the part that's speicific to your fake task), leaving you with a model which has a powerful understanding of, e.g., word meanings.

The word2vec fake task is brilliant because it requires nothing more than human-written sentences as training data--something we clearly have in abundance!

2.6. 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 the next chapter.

You may be aware that word2vec comes in two variants--the skip-gram model and the Continuous Bag-of-Words (CBOW) model. They are very similar, and I believe that understanding the material in the next chapter is more important, so I've saved CBOW for a later chapter.

You should also know that the FAQ (Chapter 5) contains additional insights into the skip-gram architecture, with answers to common questions such as:

- How do you identify multi-word "phrases" to include in the vocabulary?

- How should the output values of the skip-gram neural network be interpreted?
- Does the position of a word within the context window matter in training?

2.7. Example Code Summary

In this chapter’s Notebook, we'll reinforce our understanding of the skip-gram neural network architecture by implementing it from scratch. We'll stop at just the feed-forward implementation--that is, we'll evaluate the network on an example input word, but we won't be implementing backpropagation from scratch here.

3. Sampling Techniques

3.1. Performance Problems

In this chapter, I'll cover a few additional modifications to the basic word2vec model which are important for actually making it feasible to train.

When you read the previous chapter 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!

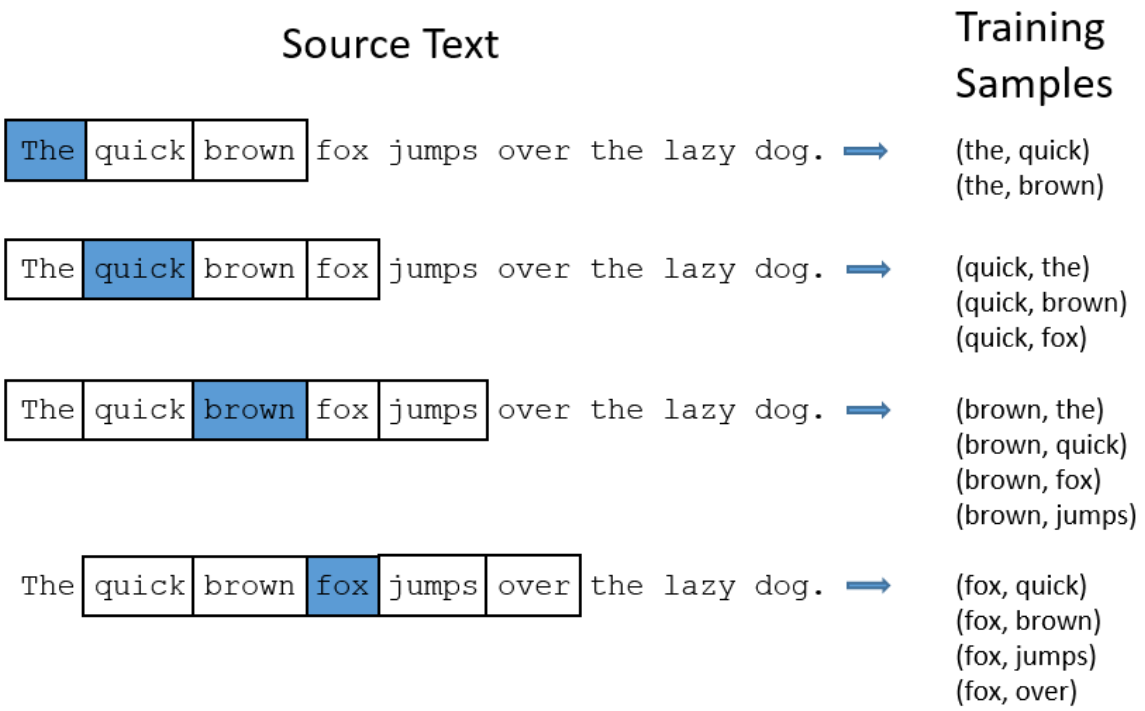
Running gradient descent on a neural network that large is going to be slow. And to make matters worse, you need a huge amount of training data in order to tune that many weights and avoid over-fitting. Millions of weights times billions of training samples means that training this model is going to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

The authors of word2vec applied a few sampling techniques to their models which both reduce the compute requirements dramatically and improved the quality of the word vectors learned. In summary:

1. They subsample frequent words to decrease the number of training examples.
2. The context window is shrunk by random amounts to give greater weight to closer context words.
3. They modified 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.

3.2. Subsampling Frequent Words

In Chapter 1, 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.

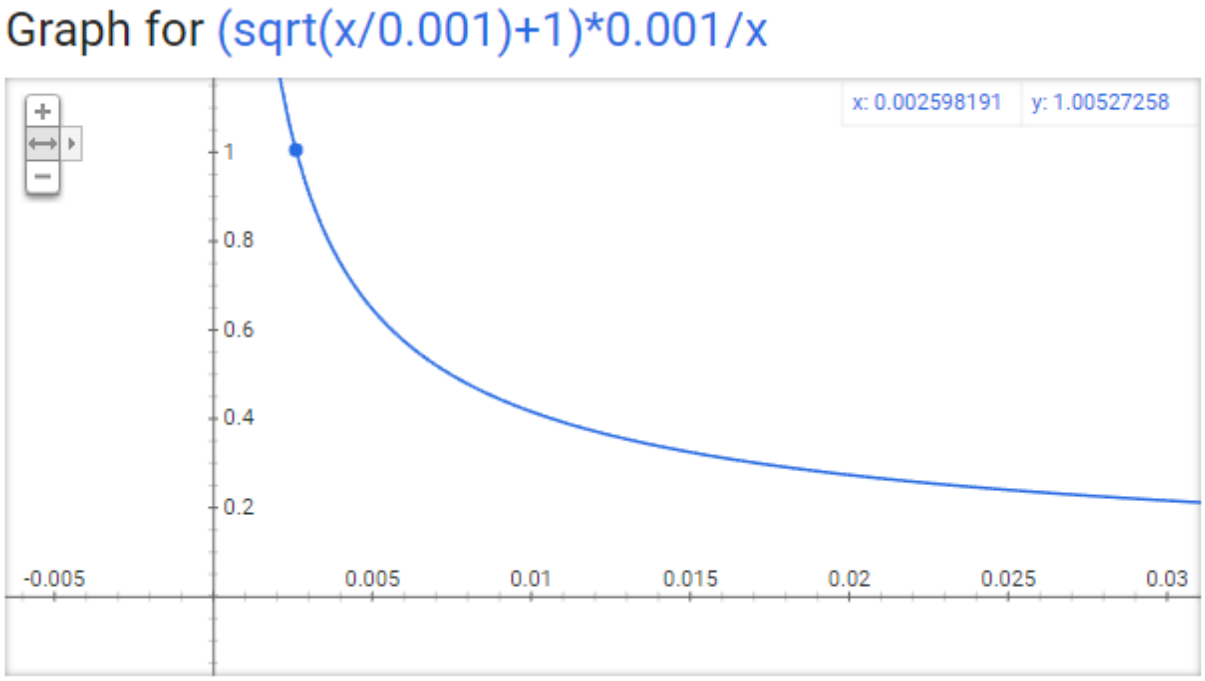
In this equation, w_i is the word, and $z(w_i)$ is the fraction of the total words in the corpus that are that word. For example, if the word "peanut" occurs 1,000 times in a 1 billion word corpus, then z('peanut') = 1E-6.

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.

$\boldsymbol{P}(w_i)$ 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 ("plot (sqrt(x/0.001)+1)*0.001/x") to see the shape.



The most interesting point on this function is the cut-off; any word which represents less than ~0.26% of the total word count *will always be kept*.

In practice, very few words are actually subsampled. In the example code for this chapter, we'll apply this function to actual word count data taken from the whole of Wikipedia. We'll see that, with the above equation, only the top 27 most frequent words would have any subsampling applied.

The impact of this is still dramatic, however, because of how common the most frequent words are. The most frequent 27 words in Wikipedia represent *33% of all the words in Wikipedia!*

Note that we are not tossing the top words, though; we are subsampling them (keeping them sometimes and tossing them others). The top 27 words represent 33% of the corpus, but based on the

sampling rates we can estimate that the number of samples will be reduced by about 22% (the example code shows how I calculated this).

Here are the top 10 words in Wikipedia along with what percent of the corpus they represent (7.8% of Wikipedia is just the word "the"!), and the probability that the above equation gives the word based on its frequency.

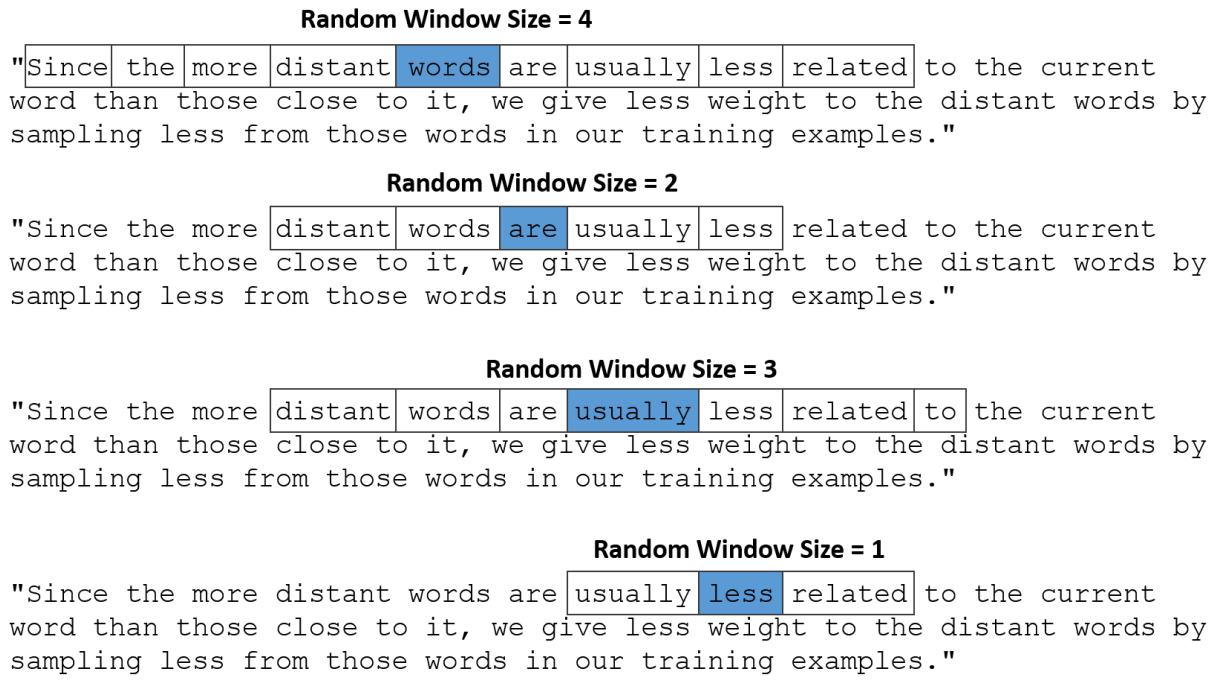
Rank	Word	Portion of Corpus	Probability to Keep
0	the	7.80%	12.60%
1	of	3.71%	19.10%
2	and	3.19%	20.84%
3	in	3.14%	21.03%
4	to	2.23%	25.66%
5	was	1.33%	34.94%
6	is	1.13%	38.63%
7	for	0.92%	43.85%
8	as	0.89%	44.80%
9	on	0.87%	45.26%
10	with	0.78%	48.68%

3.3. Context Position Weighting

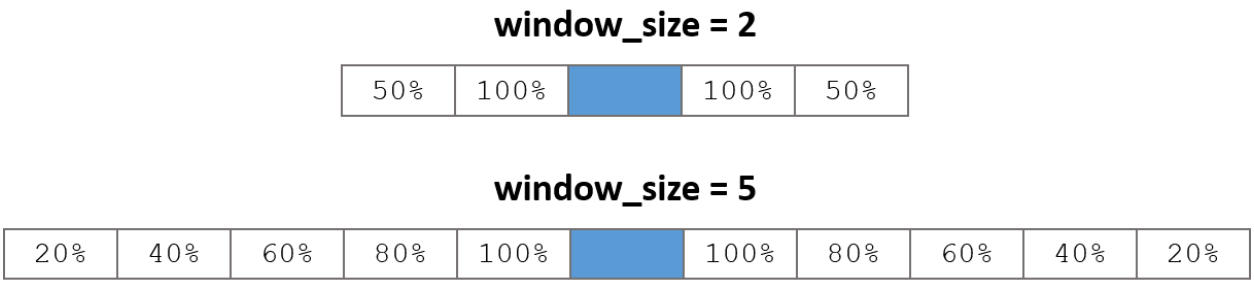
word2vec effectively weights context words differently based on their position within the context window. Or, more specifically, context words closer to the center word are given more weight than words farther out.

This is not implemented in the math or architecture of the neural network, though! Instead, this weighting is achieved by *randomly shrinking the size of the context window.*

In the word2vec C code, each time it moves the context window to the next word in a sentence, it picks a random window size in the range [1, window_size]. This means that if you train word2vec with, for example, a window size of 4, then *the actual window size will vary uniformly between 1 and 4*. The following illustration shows how this might play out with a window size of 4.



Since all window sizes in the range are equally probable, each context position has a proportional probability of being included. Here are some examples for different *maximum* window sizes; the percentage in the box is the percentage of training samples that will include the word at that context position.



Notice how the words immediately before and after are of course *always* included, and then each farther position is proportionally less common.

> You'll also find in the original C code that this random window sizing is applied to both CBOW and Skip-gram.

3.4. 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 words in the corpus. This is expressed by the following equation:

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

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 ¾ 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 word 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 $\boldsymbol{P}(w_i)$ * 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.

3.5. Example Code Summary

With both the "Negative Sampling" and the "Subsampling of Frequent Words" techniques, the sampling is based on word frequency. That is, words are sampled differently depending on how many times they

appear in the training data. In this chapter's Notebook, we will look at some actual word frequency data and see how the word2vec sampling functions behave on this data.

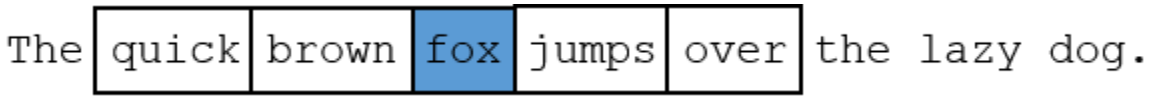
4. Model Variations

In this chapter, we'll cover an alternative to the skip-gram architecture called Continuous Bag-of-Words (CBOW), and an alternative to Negative Sampling called Hierarchical Softmax. I skipped over these variations initially since they can unnecessarily complicate your understanding of word2vec, but I am including them here for sake of completeness.

4.1. Continuous Bag-of-Words (CBOW)

CBOW is a variant of the word2vec model which essentially flips the "fake task" around. The new task we give the neural network is this: Given all the words in the context window (excluding the middle one), what is most likely the word at the center?

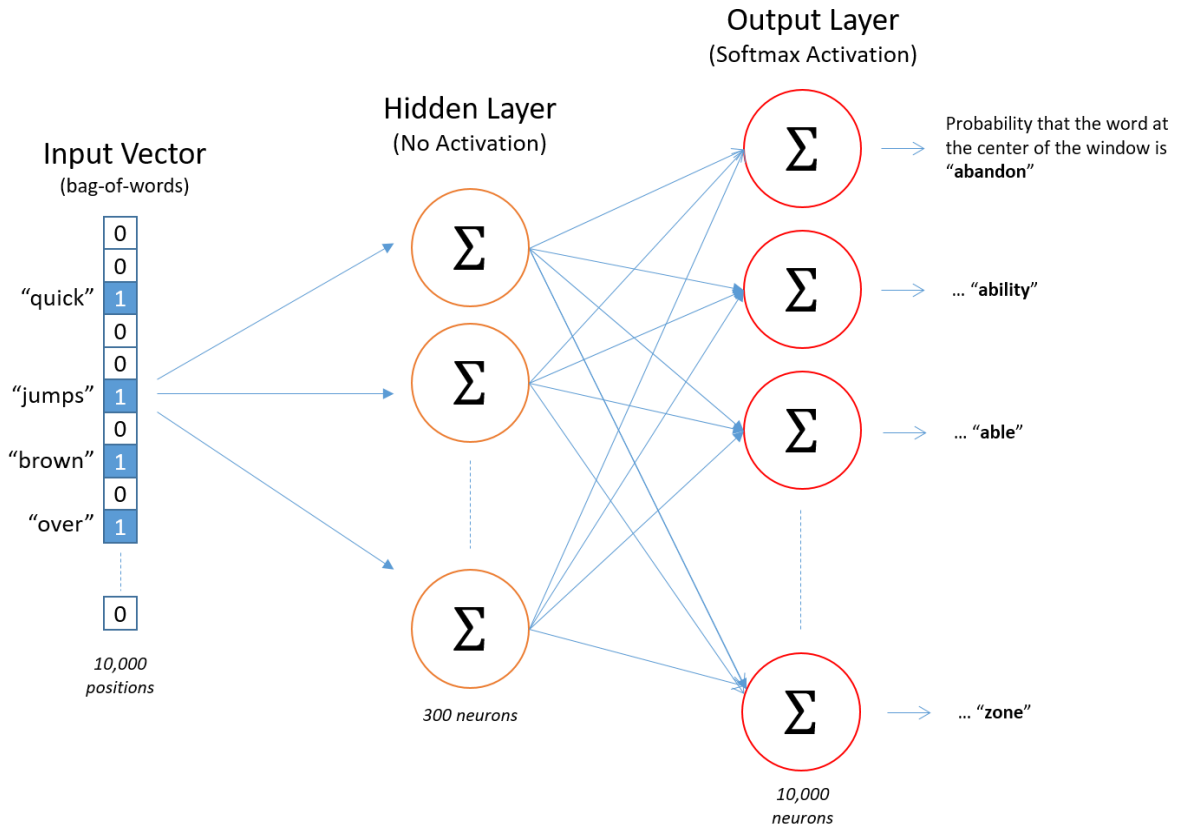
For example, say we have a window size of 2 on the following sentence. Given the words ("quick", "brown", "jumps", "over"), we want the network to predict "fox".



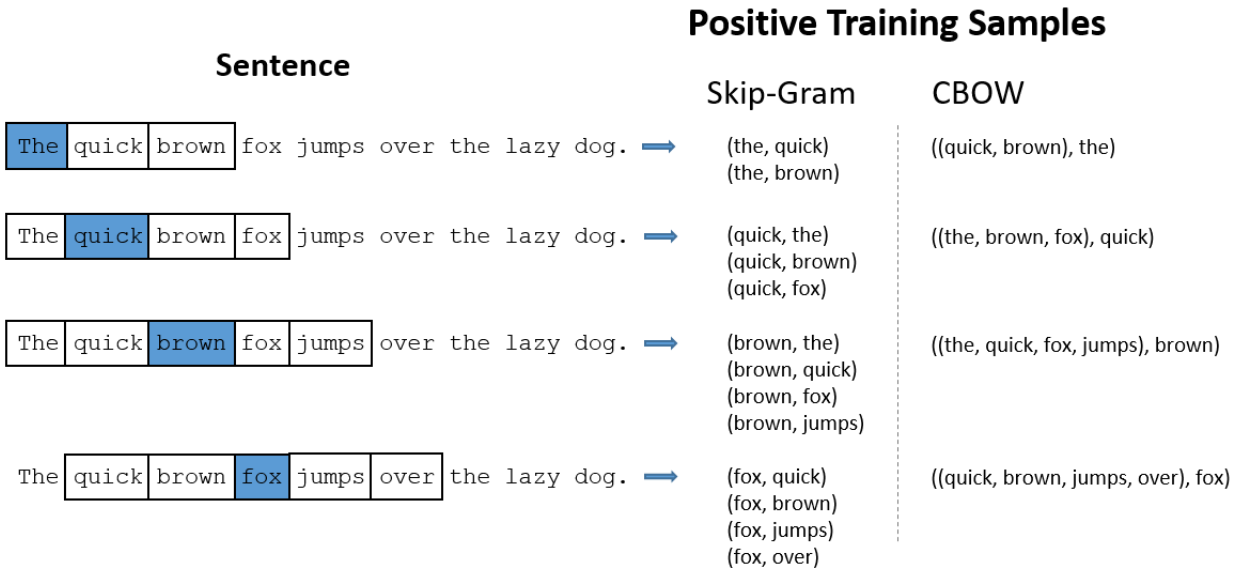
The input of the network needs to change to take in multiple words. Instead of a "one hot" vector as the input, we use a "bag-of-words"

vector. It's the same concept, except that we put 1s in multiple positions (corresponding to the context words).

The CBOW architecture then looks like the following:



The training samples for CBOW look different than those generated for skip-gram.



With a window size of 2, skip-gram will generate (up to) four training samples per center word, whereas CBOW only generates one.

With skip-gram, we saw that multiplying with a one-hot vector just selects a row from the hidden layer weight matrix. What happens when you multiply with a bag-of-words vector instead? The result is that it selects the corresponding rows and sums them together.

$$\begin{bmatrix}1 & 0 & 0 & 1 & 0\end{bmatrix} \times \begin{bmatrix}17 & 24 & 1\\ 23 & 5 & 7\\ 4 & 6 & 13\\ 10 & 12 & 19\\ 11 & 18 & 25\end{bmatrix} = \begin{bmatrix}27 & 36 & 20\end{bmatrix}$$

For the CBOW architecture, we also divide this sum by the number of context words to calculate their *average* word vector. So the output of the hidden layer in the CBOW architecture is the average of all the context word vectors. From there, the output layer is identical to the one in skip-gram.

4.2. Hierarchical Softmax

Hierarchical Softmax is an alternative to Negative Sampling. Both are methods of reducing the compute cost of training by selecting only a small number of outputs to compute gradients for.

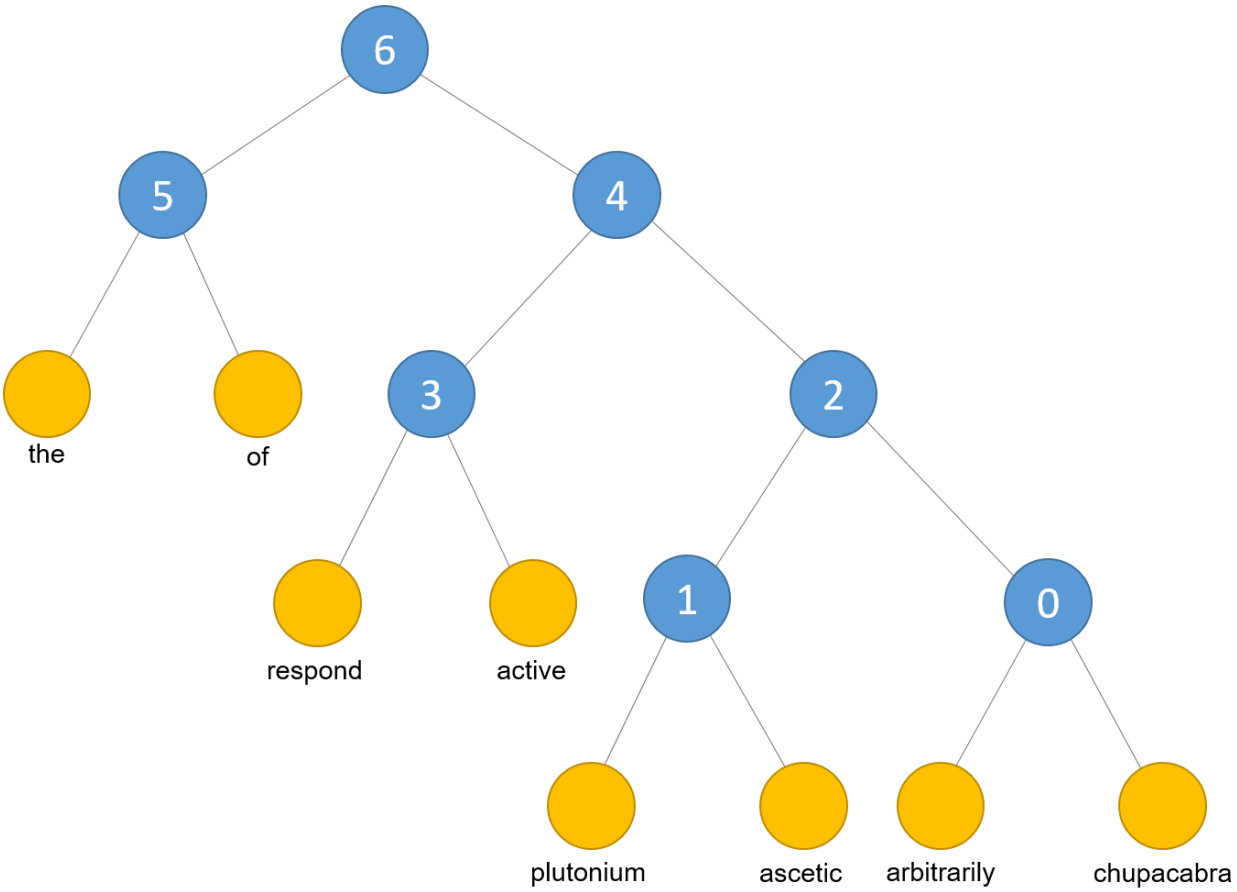
Recall that in Negative Sampling, we decide on *a set number* of negative words to use for each training input, and we choose these negative words *randomly* from the vocabulary, but with a probability related to the word's frequency.

The output layer of the Hierarchical Softmax model has a very different interpretation--the outputs of the model no longer correspond to individual words! However, on the surface, HS only differs in a few ways:

- The number of outputs we train for each training sample *varies*.
- The list of outputs to train is pre-determined.
    - For each possible context word (each word in our vocabulary), we pre-compute a short list of outputs to train.
    - Some of these outputs will be trained to output a 0 and others to output a 1.

For example, if our training pair is ("couch", "cushion"), then we go to the data structure for our vocabulary, lookup "cushion", and find there a list of, e.g., 10 specific outputs to train, along with whether each should output a 0 or a 1.

The list of outputs to train is determined in the following way. The vocabulary is organized into a type of binary tree called a Huffman tree. The words are only found at the leaves of the tree, and rarer words are found at lower levels of the tree. Below is what the tree might look like for a tiny 8-word vocabulary.



Each of the blue "internal nodes" of the tree is assigned a number. The output vectors of our model *no longer correspond to words in our vocabulary*, but rather to these blue internal nodes.

> In a binary tree like this one, if there are *n* words in our vocabulary, then there will always be *n - 1* "internal nodes" in our tree.

When we build this tree, we also record the path to each word. For example, if you looked up the word "active" in our vocabulary, our vocabulary data structure would give you the path (6, 1), (4, 0), (3, 1);

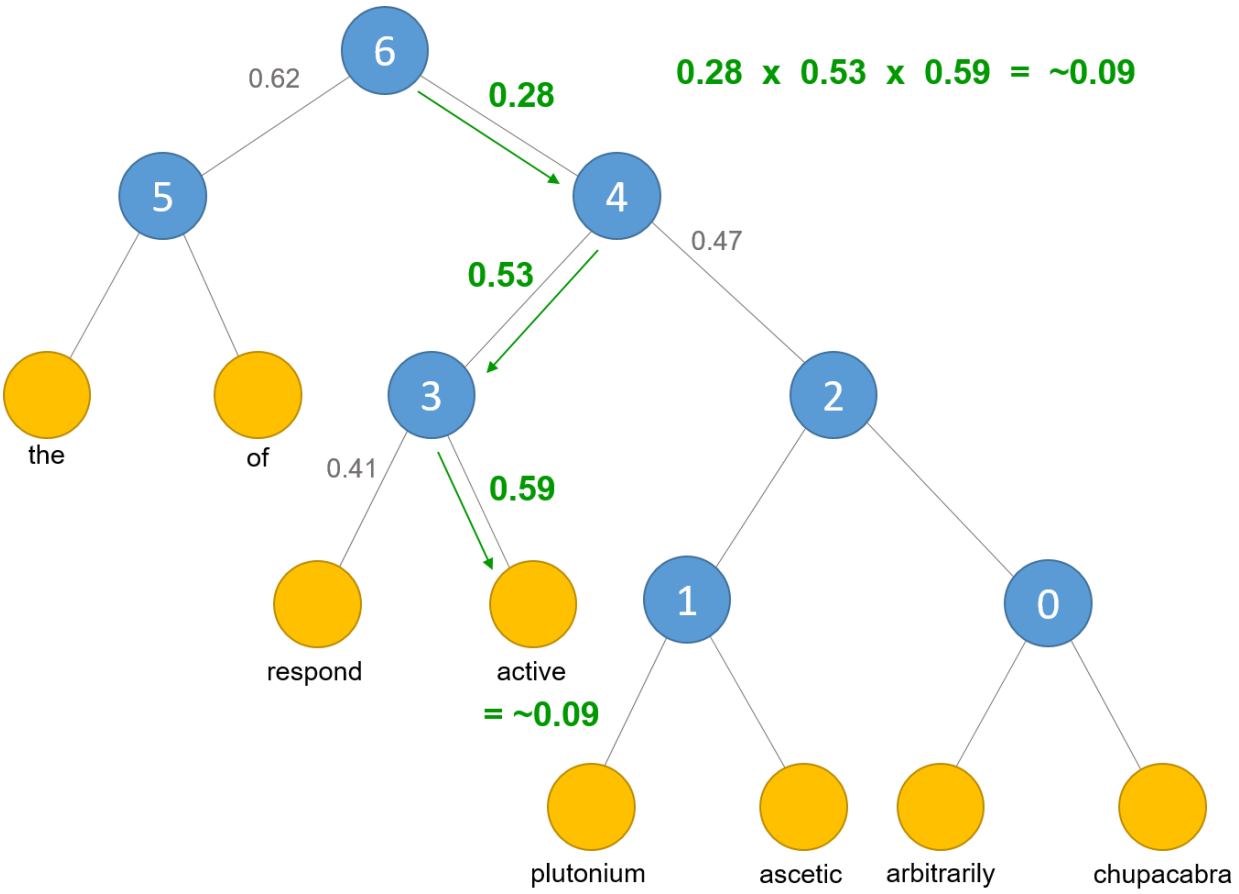
meaning, take the right branch (“1”) at node 6, take the left branch (“0”) at node 4, and finally take the right branch (“1”) at node 3.

If we had the training pair (“chupacabra”, “active”), then we would train the outputs according to the path to “active”. That is, we would train node 6 to output 1, node 4 to output 0, and node 3 to output 1.

Kind of bizarre behavior--what is this model learning? It’s actually learning an approximation to the Softmax function.

Let’s say your input word is “chupacabra”, and you want to know the probability of the context word being “active”. Here’s an example of you would calculate this using our model.

1. Take the input word vector for “chupacabra”, and take its dot product with output vectors 6, 4, and 3, and apply the sigmoid activation to each. Let’s say this yields the values 0.28, 0.47, and 0.62.
    a. Each of these values represents the probability of taking the *right-hand branch* at that node!
2. To get our final output probability, we multiply together the probabilities for taking the branches that will lead us to “active”.
    a. Specifically, (0.28) x (1 - 0.47) x (0.59) = ~0.09



If we repeated this process for every context word in the vocabulary, all of their probabilities would sum up to 1.0! To see why that's the case, notice how each node is just breaking up the total probability into smaller and smaller chunks.

The tree learned these probabilities based on the number of times we chose each path in the tree. We could read the tree in the following way:

Over the course of training, whenever the input word was "chupacabra"...

- 28% of the time, the context word was found under node 4, and the other 62% under node 5.
- If we made it to node 4, then 47% of the time the context word was found under node 2, and the other 53% under node 3.

- If we made it to node 3, then 59% of the time the word was "active" and 41% of the time the word was "respond" .

Of course, it won't be perfect, since our training data could never contain every possible combination of input and output word, and because this tree structure (and the output vectors it's learning) are *shared* across every input word.

That means that, once again, if the model needs to output very similar probability distributions for two words, then it's best bet is to learn *similar input word vectors* for those two words!

4.3. Practical Differences

skip-gram vs. CBOW

When you compare the training samples generated by skip-gram and CBOW, it's clear that skip-gram generates many more samples. The number of samples generated by skip-gram depends on the window size, whereas for CBOW there is only one sample per word in the sentence.

This leads to a number of practical differences:

- Skip-gram takes much longer to train because it produces many more training samples.
    - The larger the window size, the more samples it will create and the longer it will take to train.
- Skip-gram may perform better when the training set is small. It generates more training data from the same amount of text.
- Skip-gram may create better representations for rare words. Since a rare word by definition doesn't appear much in the corpus, skip-

gram makes the most use out of the limited number of samples (by generating more training samples than CBOW).

I think it's interesting to note that If you train a word2vec model in *gensim*, the default model choice is CBOW.

There are a few other observations which Mikolov offered here. I don't have an intuitive explanation for these, but I'll pass them along.

- CBOW performs "slightly better" than skip-gram at capturing syntactic relationships (i.e., recognizing different conjugations of a verb like run, ran, running, runs)
- CBOW performs "slightly better" at representing the more frequent words.

So, in summary:

	Which handles it better?	
	CBOW	Skip-gram
Small Datasets		✔
Training Speed	✔	
Rare Words		✔
Frequent Words	✔ "slightly"	
Syntactic Similarity	✔ "slightly"	

Negative Sampling vs. Hierarchical Softmax

I haven't heard, or been able to figure out myself, the rationale for the differences in performance between negative sampling and HS. Nevertheless, I can share some empirical observations that I've made as well as those from the original authors.

First, in my experience, hierarchical softmax takes about 50% longer to train than negative sampling when using the default model parameters.

Second, you can find some comments on the differences at the homepage for the original word2vec C library: https://code.google.com/archive/p/word2vec/. They claim that HS handles rare words better, whereas negative sampling handles frequent words better. Also, they've found that negative sampling is better if you are restricted to small vector sizes. Smaller vector sizes mean a lower memory footprint and faster execution of your downstream task--if your application has strict performance requirements, then negative sampling may help.

In summary:

	Which handles it better?	
	Negative Sampling	Hierarchical Softmax
Training Speed	✔	
Rare Words		✔
Frequent Words	✔	

Smaller Vectors	✔	

5. Backpropagation Training

In this chapter, we'll go into more depth on the word2vec model training by walking through some examples and showing the specific steps taken to update the weights.

For this discussion, I will be using the original word2vec C code as the authority for how things are implemented, rather than the papers.

This leads to a couple important changes from what's been presented in the previous sections.

5.1. Matrix Names

First, we'll do away with the unnecessary one-hot input vector. Previously, we had an input layer (the one-hot vector), a hidden layer weight matrix, and an output layer weight matrix. Now we'll simply call these the Input matrix and the Output matrix.

We also don't need the term "neuron" anymore--it will serve us better to talk in terms of word vectors.

The Input Matrix is our word vector lookup table—it has one row per word in the vocabulary. The Output Matrix is the same size, and is just a necessary piece of our "fake task" (predicting context words).

5.2. Output Activation

In our initial formulation of the skip-gram neural network architecture, we said that the output layer uses a softmax activation function:

$$S\left(x_i\right) = \frac{e^{x_i}}{\sum_{i=1}^{n} e^{x_i}}$$

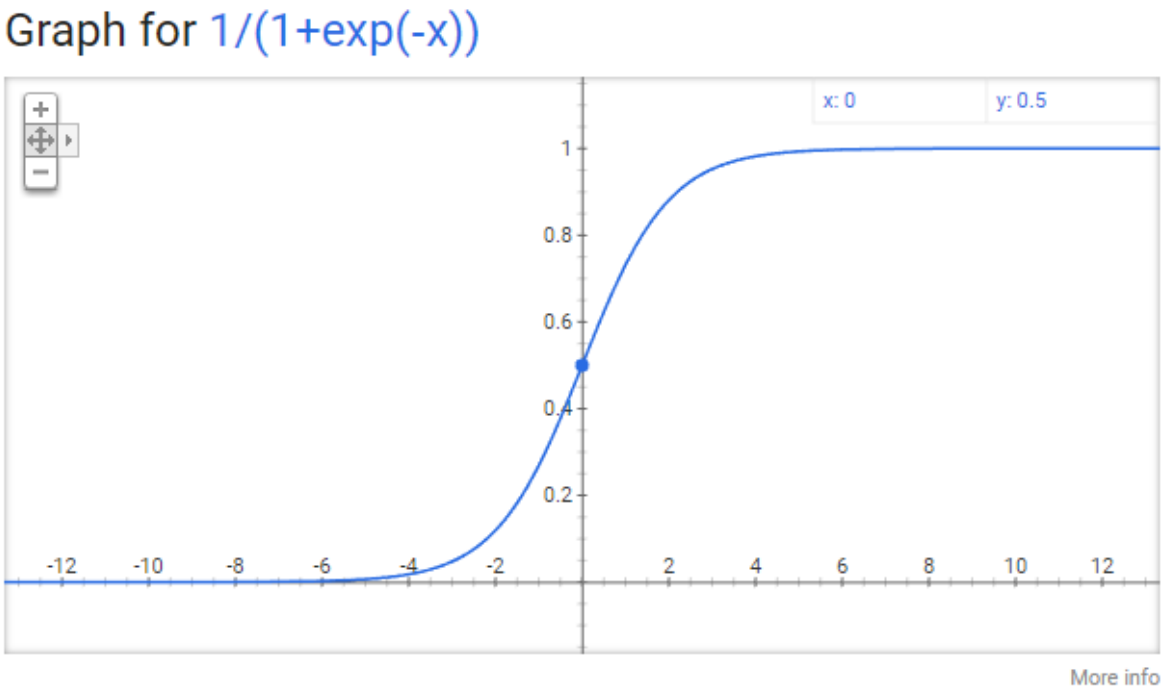
This follows the description in the authors' *first* paper, where the model was presented without Negative Sampling.

In the word2vec.c implementation (and as explained in the *second* paper), however, they instead use the sigmoid activation function:

$$g(x) = \frac{1}{1 + e^{-x}}$$

You can throw this into Google to see what it looks like:

```
plot y = 1 / (1 + exp(-x))
```



This change is explained in the authors' second paper in their discussion of the negative sampling approach: "Thus the task is to distinguish the target word w_O from draws from the noise distribution $P_n(w)$ using **logistic regression**, where there are k negative samples for each data sample." (emphasis added).

5.3. Skip-gram with Negative Sampling

In this section, we'll walk through the weight updates for a skip-gram model with negative sampling, as performed in the `TrainModelThread` function in `word2vec.c`.

We're going to walk through an example to help illustrate the points. The words and values used in this example are taken directly from an actual training session.

In our example, we are working on the training sample ("thought", "well") which came from the following sentence in our dataset:

```
              -3         -2       -1         input       +1       +2         +3

"your   (questions)   (are)   (well)   [thought]   (out)   (and)   (reasoned)"
```

For negative sampling, we also need to pick five random words (using the modified unigram distribution, covered in the negative sampling chapter) to use as negative samples. These are: 'busting', 'significant', 'england', 'together', and 'arbitrarily'.

Initial Weight Values

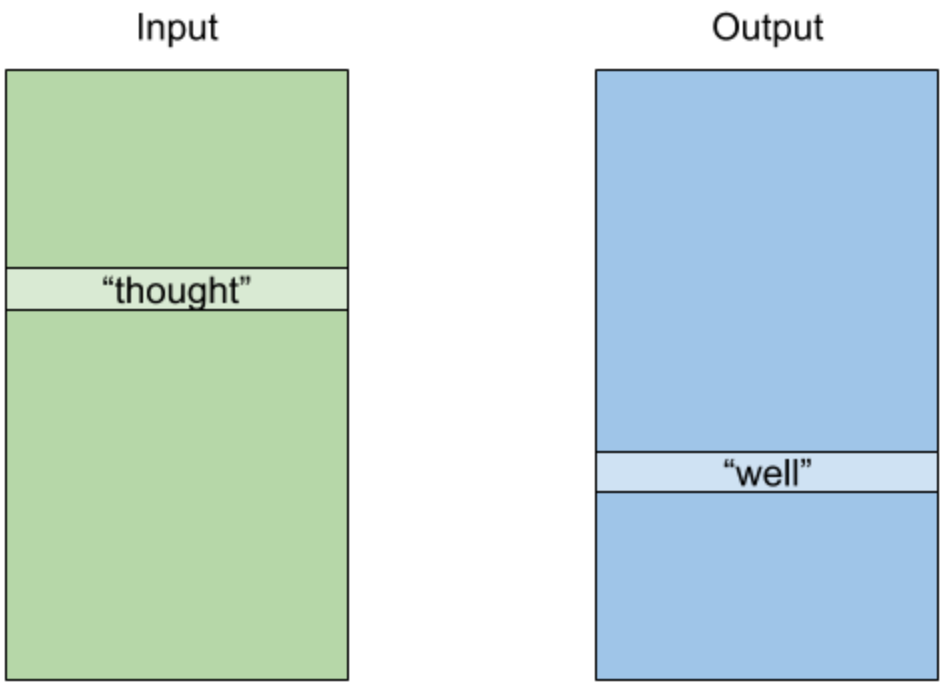
At the very beginning of training, all weights in the Output Matrix are set to 0, and all weights in the Input Matrix are initialized to random values. You can see this in the `InitNet` function in `word2vec.c`.

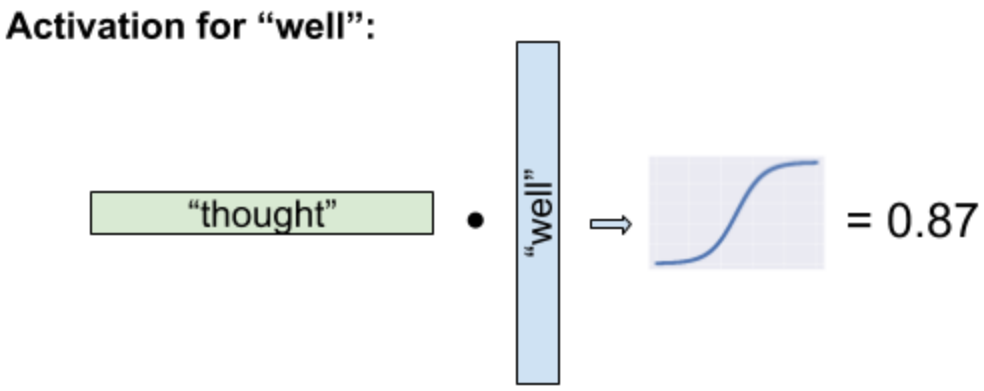
For our walkthrough, I'm actually using weights from a *partially trained* model--the weights have already gone through two training passes. This way, the values are fairly sensible, but there's still room for improvement.

Forward Pass

Let's start with the positive sample: "thought", "well"

The first step in backprop is to run a forward pass of the network. To do this, we select the vector for "thought" from the Input Matrix, and the vector for "well" from the Output Matrix. We take the dot product of these two vectors, and apply the sigmoid activation function to get an output value of `0.87` (I will also refer to this as the "activation value" for "well").





Remember that the output of the network *is not a measure of word similarity*. Rather, it reflects how likely you are to find the word "well" in the vicinity of "thought".
As an example of the distinction, consider how the words "thought" and "think" are very similar in meaning, but are unlikely to appear close together.
The similarity of the input word vectors for “thought” and “think” is high, but the network output value for “think” is low.

The next step in backprop is to calculate the error. The error is simply the difference between the true label (1.0 for positive samples and 0.0 for negative samples) and the output. For simplicity, we're also going to fold the learning rate ☒ ("alpha"), into the error value. The learning rate defaults to 0.025.

> The learning rate is a parameter which significantly reduces the magnitude of our weight updates to ensure that we don't make too large of adjustments.

Here are the actual activation values and errors for our one positive sample and our five negative samples.

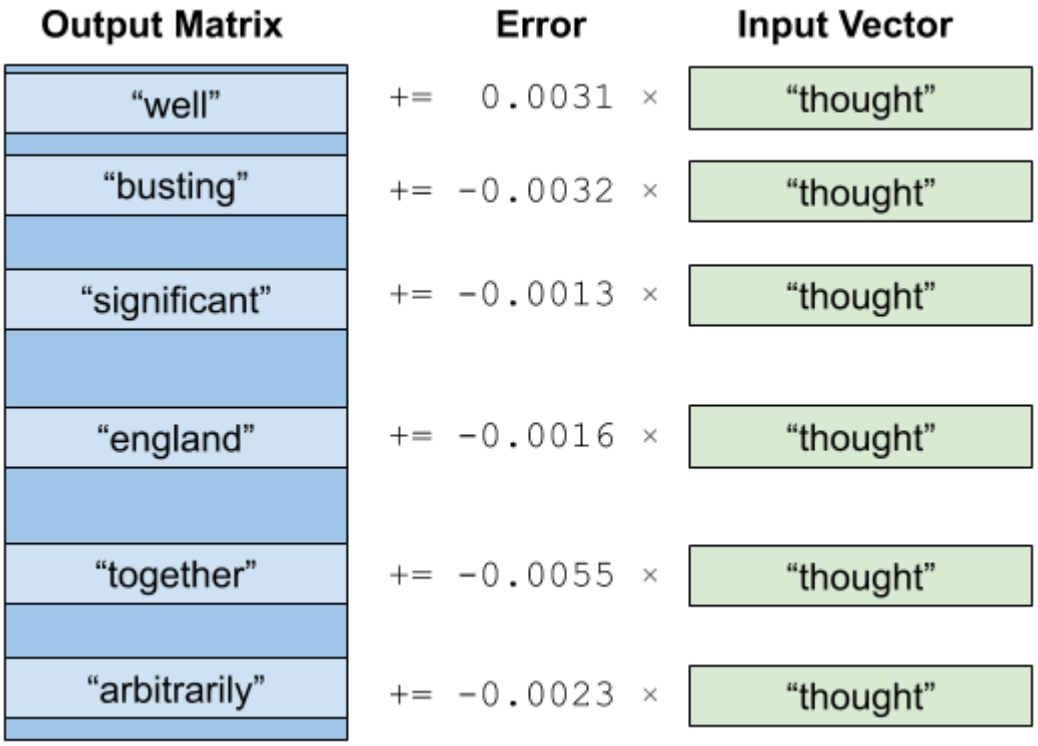
Input	Output	Label	Activation	Error
thought	well	1	0.8748	0.0031
thought	busting	0	0.1274	-0.0032
thought	significant	0	0.0512	-0.0013
thought	england	0	0.0645	-0.0016
thought	together	0	0.2206	-0.0055
thought	arbitrarily	0	0.0910	-0.0023

Output Weights Update

The next step after calculating the error is to update the output weights.

To update the output weights for "well", we take the input vector for "thought" and multiply it by the error `0.0031`, and this gives us our **gradient** for the output weights for "well". (gradient is the term used to refer to the amount we update the weights by).

The following illustration shows this step being performed for each of the output words.



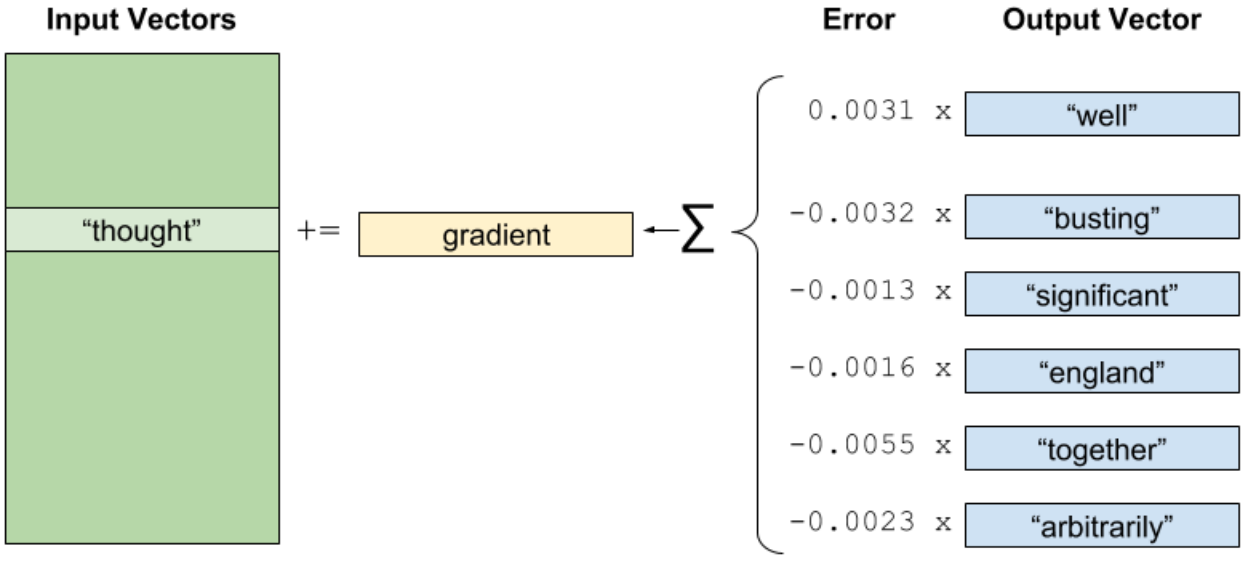
The "+=" sign means update the left hand side by adding the right hand side to it. For example:

`a += 1` is the same as `a := a + 1`

Input Weights Update

After updating the output weights, we continue backward to update the input weights.

This update is (almost) the reverse of the output weights update. To update the input weights for "thought", we take the output vector for "well" and multiply it by its error of `0.0031` to calculate a gradient. We repeat this for all 6 output words and sum them together for the final gradient vector for "thought". The input vector for "thought" is updated by simply adding the gradient vector to it.



The order of the weight updates is performed such that the current changes to the weights don't impact any of the current error calculations.

For each ouput word:

1. Calculate the output error for the current word.
2. Accumulate the input vector gradient, but don't apply it.
3. Update the output vector weights for the current word.

Finally, update the input vector weights.

New Output Values

After making all of the above adjustments to the weights, we can run another forward pass to see how the output values have changed. We'll find that the output of the network for "well" has moved slightly closer to 1.0, and the outputs for all of the negative words have moved slightly closer to 0.0.

Input	Output	Label	Previous Activation	Error	New Activation
thought	well	1	0.8748	0.0031	0.9089
thought	busting	0	0.1274	-0.0032	0.0917
thought	significant	0	0.0512	-0.0013	0.0440
thought	england	0	0.0645	-0.0016	0.0537
thought	together	0	0.2206	-0.0055	0.1294
thought	arbitrarily	0	0.0910	-0.0023	0.0711

5.4. Example Code Summary

In this chapter's Notebook, we'll get hands on with backpropagation and implement the weight updates (for a skip-gram model with negative sampling) from scratch!

6. Subword Vectors & fastText

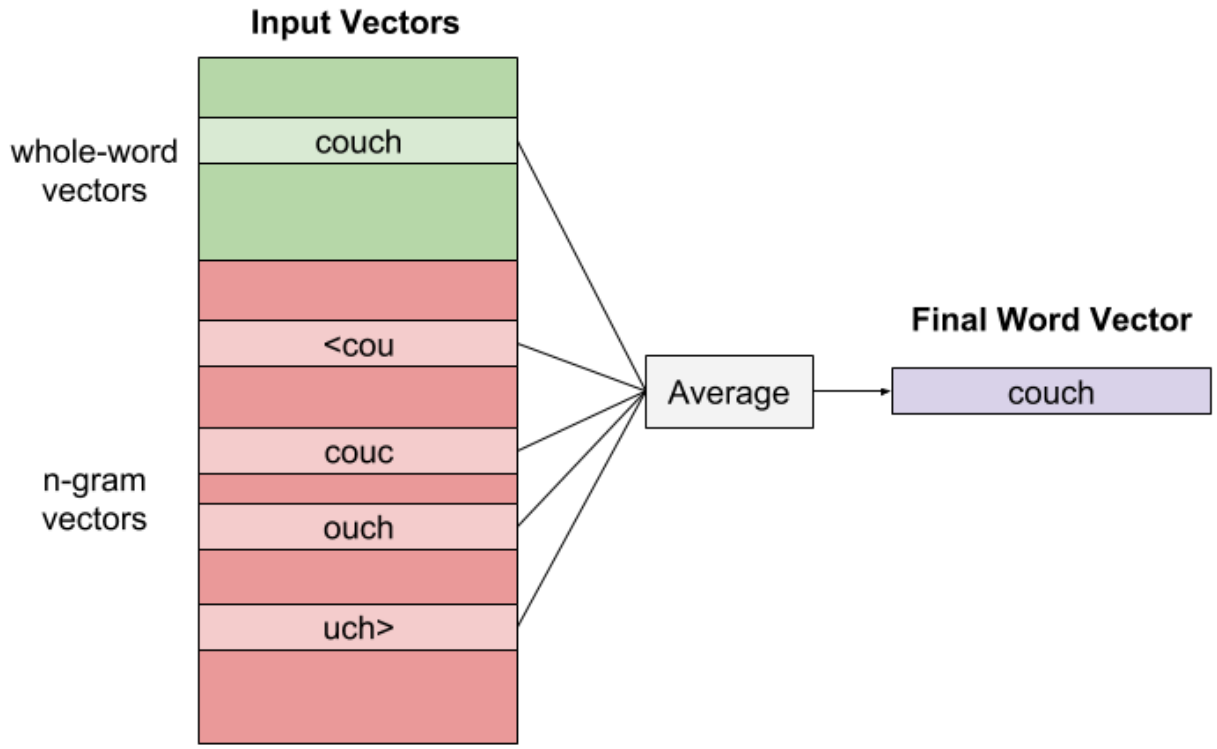
6.1. Overview

So far, we have been looking at applying word2vec only for whole words or multi-words like "computer_science".

In this chapter, we'll look at what happens if you learn vectors for both whole words and "subwords". The subwords are created by breaking down a word like "computer" into all possible **character n-grams**, like "comp", "ompu", "mput", etc.

> Side Note: The term n-grams can be used both to refer to "character n-grams" and "word n-grams", so it helps to clarify which type you're dealing with.

To create a vector for the word "couch", we take the average of the vector for "couch" and the vectors for all of its the n-grams.



The main motivation for incorporating these subword vectors is that they can help us build decent word vectors *with less training data*.

For most applications, it's best to train the word model on your own text data rather than something generic like Wikipedia. Application-specific training data is often much smaller than Wikipedia, though, so being able to learn better representations from small datasets is an excellent strength.

Learning from less data is also valuable for learning good representations for **rare words.** Rare words are ones which aren't used frequently and which we therefore don't have many training samples for. Don't just think of these as "bizarre" words, though; a rare word could also be a misspelling or an unusual form of a more common word.

The trade-off of using subwords is a higher compute cost during training and a larger vector matrix (containing all of the whole words along with an even larger set of subwords).

Subwords also don't appear to improve the representations for **common words** (words which are already well represented in the training data) and may even lead to slightly *worse* representations for these.

As a reader, the good news I have for you is that incorporating subword information is primarily a change in how we interact with the training text and our vocabulary. Otherwise, the neural network architecture and the sampling techniques that we learned about in the previous chapters are essentially unchanged!

6.2. History

Tomáš Mikolov authored word2vec while part of the Google Brain team in 2013. In 2014 he left Google for Facebook's AI research lab (FAIR). You can view his FAIR profile here.

At Facebook, Mikolov was involved in further word2vec research, and published the paper Enriching Word Vectors with Subword Information in 2016. Along with this paper came a new implementation of the word2vec code which supported subword information, a C++ library called fastText.

While the original word2vec C code seemed primarily intended as sharing the source code for the paper, fastText is presented more as a proper open-source library.

Technically, "fastText" is a new C++ implementation of the word2vec model which supports subword information, not the name of a new algorithm. It's convenient, however, to conflate the two and use "fastText" as the name of the algorithm as well. For instance, gensim now also has a "fasttext" word model which supports subword information. The unfortunate consequence is that it gives the impression that fastText is a complete departure from (or replacement for) word2vec.

FAIR also made a splash by publishing pre-trained word models for many languages on both Wikipedia and the Common Crawl.

Together, I think the release of fastText and these pre-trained models have inspired two common misconceptions:

Misconception	Reality
word2vec is outdated now, you should be using fastText.	fastText is an extension of word2vec, not a replacement, and has pros and cons. The original word2vec may still be the better fit for your application.
No one needs to train their own word model anymore. Facebook has already built the best possible one, so just use that.	Quite the opposite--most likely you should be training your own model on your own dataset to learn the right word meanings in the context of your application.

> From the fastText paper - "In general, when using vectorial word representations in specific applications, it is recommended to retrain the model on textual data relevant for the application."

With that out of the way, though, we'll see in this chapter how subword information may in fact be greatly valuable in your application.

6.3. Extracting n-grams

For this subwords approach, we choose a range of n-gram lengths to extract. The default is to extract all n-grams between 3 and 6 characters.

Before extracting n-grams from a word, we surround the word with carrots: '<' and '>'.

So the word `kaggle` becomes `<kaggle>`

With the default range of n-gram sizes, the word "kaggle" produces 18 n-grams:

3-grams	`<ka`	`kag`	`agg`	`ggl`	`gle`
4-grams	`<kag`	`kagg`	`aggl`	`ggle`	`gle>`
5-grams	`<kagg`	`kaggl`	`aggle`	`ggle>`	`le>`
6-grams	`<kaggl`	`kaggle`	`aggle>`		

Adding the carrots around the word accomplishes a number of things:

- It differentiates n-grams at the beginning and end of a word from those in the middle.
- It differentiates n-grams representing a whole word from subwords.

6.4. Benefits of n-grams

Let's look in more detail at the different ways that n-grams can improve our word representations.

Morphology

A given word may have multiple forms which have different spellings but share their meaning. For example, "flabbergast", "flabbergasts", "flabbergasted", and "flabbergasting" are all different conjugations of one verb.

This unusual word is most often used in a statement like "I was flabbergasted when...", meaning I was really surprised or astonished. Based on a Wikipedia dump from March 2019, here are the total number of occurrences of these different conjugations:

Form	Count
flabbergast	10
flabbergasts	0
flabbergasted	218
flabbergasting	12

It's clear that the other three forms are very under-represented. However, because they all share most of their n-grams, the subword information learned for "flabbergasted" will help to improve the vector representations of the others.

Out-of-vocabulary words & misspellings

If we encounter a word in our application that did not appear in our training data, we won't have a vector to represent it (at least not without applying some kind of clever trick). Subword information gives us a decent way to represent these **out-of-vocabulary (OOV)** words. We simply chop the OOV word into all of its n-grams, look up the vectors for those n-grams, and average them together to give us the word vector.

If the OOV word is just something bizarre, then this technique may not buy you much. For instance, you're not going to figure out that "kaggle" is a data science competition website if it didn't appear in your training data and you're just combining its n-grams!

A more practical situation, however, would be handling misspellings. From the same Wikipedia dump, the word "kaggle" appears in Wikipedia 55 times, but its misspelling "kaggel" appears 0 times.

Again, the OOV word might also just be a rare conjugation or word form like "flabbergasts" (occurs 0 times in Wikipedia) or "Kaggler" (meaning someone who participates in Kaggle competitions).

> What if an OOV word includes an n-gram that wasn't in our training data? We'll return to this at the end of the chapter, but in short, this is handled through hashing. The n-grams are hashed into buckets, and there is one "n-gram vector" per bucket. There is no collision handling, so multiple different n-grams may hash to the same bucket / vector! Hardly fool-proof, but it apparently works well enough in practice.

Cons

Keep in mind the disadvantages, however:

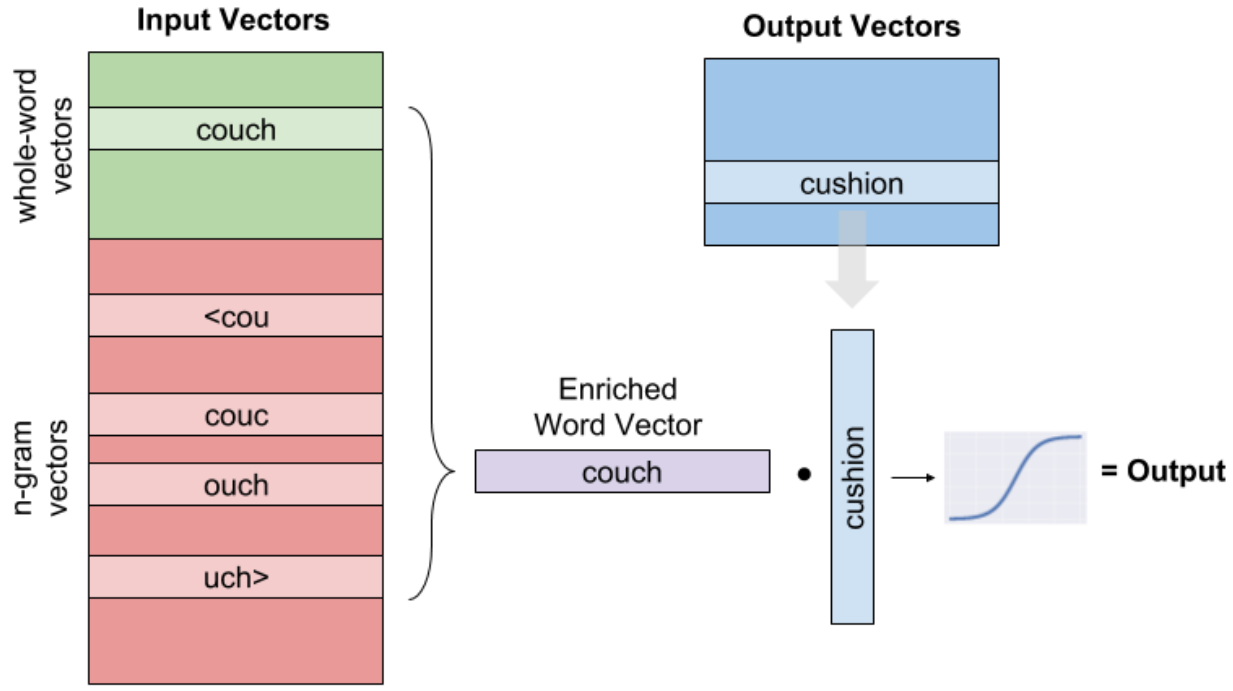
- The word vectors for common words aren't necessarily improved by the subword vectors and may even be slightly poorer.
    - The authors ran a word vector benchmark which uses mostly common words, and they found that incorporating subword information decreased their performance slightly compared to normal word2vec (see the scores for the English WS353 dataset in Table 1 on page 5 of the paper. Word2vec scored 73 and fastText scored 71).
- The number of vectors we need to learn grows substantially, which means the training time and memory requirements for a given dataset size are higher.
    - In the example code, the word2vec model took 43s to train on my desktop, while fastText took 268s (4min 28s), which is ~6.2x longer.

6.5. Training with n-grams

Training the network with these added n-grams is less different than you might expect. This is because the n-grams are only present in the Input Matrix and not the Output Matrix. For the skip-gram architecture, we still only train the network on pairs of words. We are *not* training on pairs of n-grams (for example, we do *not* train ("<cou", "<cus"))

Feed Forward Operation

Below is an example of the skip-gram architecture run for the input word "couch" and the output word "cushion".



The input vector for "couch" and all of its n-grams (the illustration only shows 4-grams to save space) are averaged together to form a new vector. This vector has been "enriched" with subword information, so I'll refer to it as the **enriched vector** for "couch". The model output is

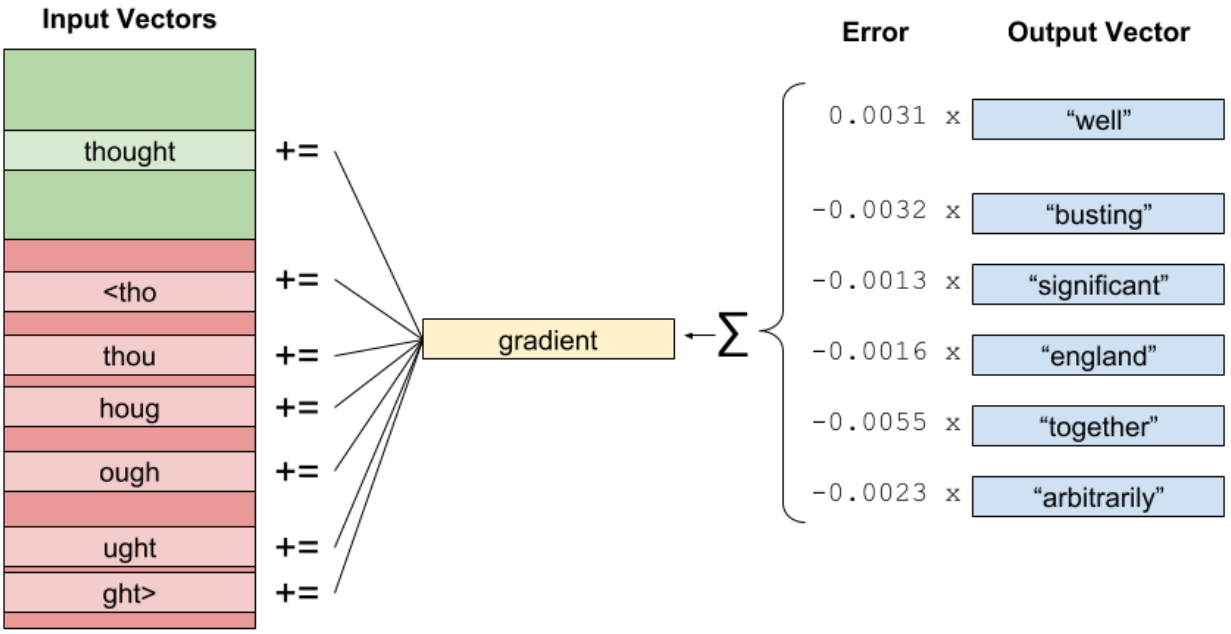
just the dot product of the enriched "couch" vector and the output vector for "cushion", followed by the sigmoid activation function.

Backpropagation Training

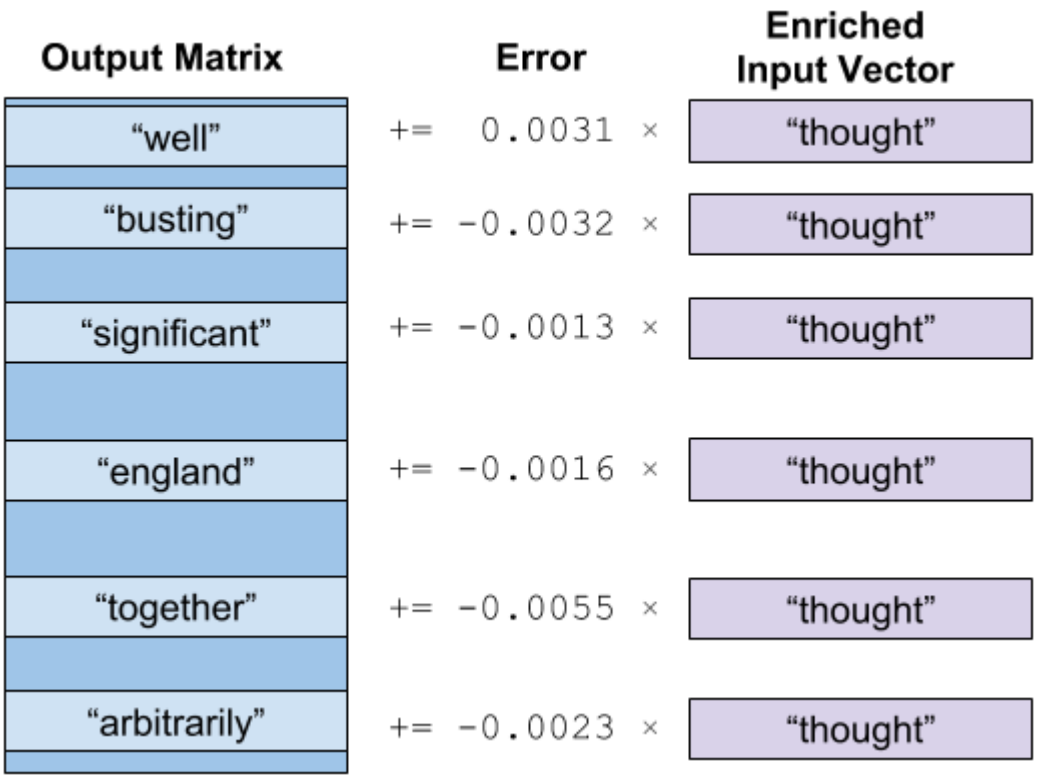
The weight update is only slightly different than standard word2vec.

I'll re-use the ("thought", "well") example from Chapter 5 here to illustrate the difference, so let's review some of the details. In this example, "well" is the positive sample and the other five output words ("busting", "significant", "england", "together", "arbitrarily") were selected as the negative samples. The error for each output word was calculated as the difference between the label (1 or 0) and the network output for that word.

For updating the input vectors, we first accumulate the gradient vector by multiplying each output word's error by it's output vector, and summing these all together. This gradient is then applied to both the whole word vector *and all of its n-gram vectors*.



The process for updating the output weights is also nearly unchanged--the only difference being that we use the enriched version of the input vector.



6.6 Hashing n-grams

There is an interesting implementation detail in fastText around how the n-gram vectors are handled, which we touched on briefly in section 6.4.

Recall that the Input Matrix now consists of two sections. The first section is for whole-word vectors and the second is for n-gram vectors.

In the whole-word vector section of the matrix, each row corresponds to a specific word, and there is a 1:1 relationship between words and rows. The model has a list of all words in the vocabulary, and has a mapping from word string to row number. The number of rows for whole-word vectors corresponds exactly to the number of words in the vocabulary.

The n-gram rows are handled differently. Unlike the vocabulary of "words", there is no vocabulary of n-grams. Instead, there is a fixed number of n-gram rows (the default is 2M). When working with a particular n-gram string, like "`<kag`", we hash the string to one of these 2M rows, and that is the row that we use for that n-gram. fastText does not do anything to prevent n-gram collisions, though! Multiple distinct n-grams may map to the same row.

In the same vein, some of these 2M rows may be unused, meaning they are unmodified during training, and just contain the random values they were initialized with.

There is also no "n-gram vocabulary" created where we can check "Did we learn an n-gram for `<asdf`?". We just hash `<asdf` onto one of the n-gram rows, and whatever is there is what we use, whether it's garbage or not.

Selecting the number of n-gram "buckets" to use will therefore be a trade-off between matrix size and number of collisions (fewer rows means more collisions).

6.7. Example Code Summary

In this chapter’s notebook, we will train a word2vec model with subword information (using the ‘fasttext’ model in gensim) on the Wikipedia Attack Comments dataset. We'll look at how the training time and memory requirements compare, as well as the quality of the resulting vectors.

A. Bonus #1 - FAQ

A.1. What are the explanations for the names "Continuous Bag-of-Words" and "Skip-gram"?

A.2. How do you build a vocabulary that includes multi-word names and phrases?

A.3. Does the position of a word within the context window matter in training?

Does the network output a different probability distribution for each context position?

A.4. How can the probabilities sum to one if certain words always appear together?

A.1. What are the explanations for the names "Continuous Bag-of-Words" and "Skip-gram"?

Continuous Bag-of-Words

In Natural Language Processing, a "Bag-of-Words" vector is a representation of a piece of text (such as a document) which has one position for each word in the vocabulary, and is populated with the counts for each word in the document. For the CBOW model, the input vector, which stores the counts of the words in the context window, is a "Bag-of-Words" vector.

The first word2vec paper talks about learning "continuous word vectors". This simply refers to the fact that the word vector features are continuous variables--that is, floating point numbers rather than integers. I believe this is the rationale behind the word "Continuous" in the model name.

Skip-gram

In Natural Language Processing, a *word n-gram* (as opposed to a *character n-gram*) refers to taking 'n' adjacent words together as one, such as "abraham_lincoln", which is a "bigram".

Recall that the skip-gram model uses pairs of words as it's training sample--the input is the word at the center of the context window, and the output is any one of the other words in the context window. Most of these word pairs are not proper bi-grams, though, since the words in the pair aren't adjacent--instead, most of the pairs involve "skipping over" other words.

Note that the paper actually calls this the "Continuous Skip-gram Model" (the title of Section 3.2 in the first paper), but most discussions of the model (mine included) omit the extra word.

A.2. How do you build a vocabulary that includes multi-word names and phrases?

In their second paper, the word2vec 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 second paper. They shared their implementation in word2phrase.c--I've shared a commented (but otherwise unaltered) copy of this code here.

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.

A.3. Does the position of a word within the context window matter in training?

(And if so, does the network output a different probability distribution for each context position?)

This is a common source of confusion, because the answer to the first question is "Yes, word2vec does take context position into account", while the answer to second question is "No, the neural network does not have different output weights for different context positions".

word2vec takes into account the distance between a context word and the input word, such that the words immediately before and after the input word are given higher weight than the words two positions away, which in turn have higher weight than the words three positions away, and so on.

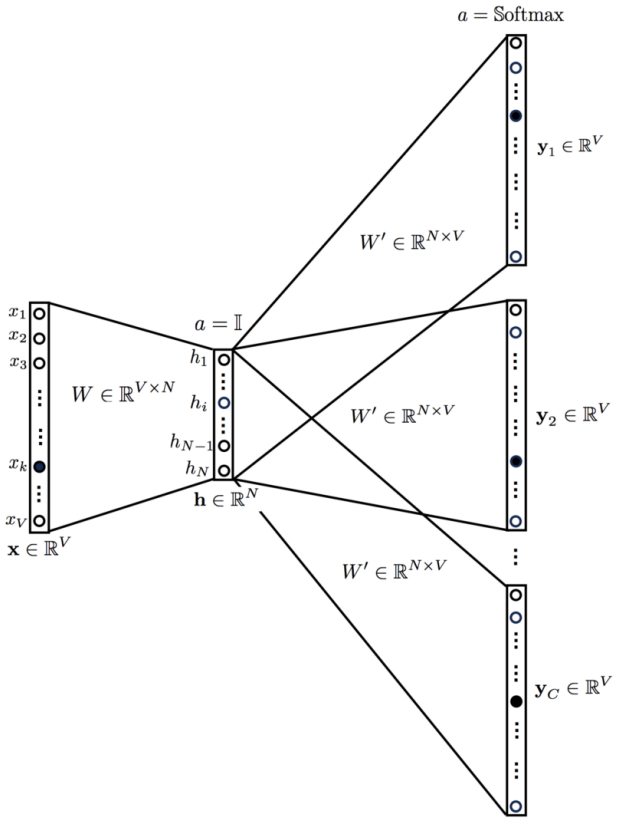
This is not implemented in the formulation of the neural network, though, but rather through how training samples are selected.

In the word2vec training code, the window size "C" is actually shrunk by a random amount, such that the window size varies uniformly between 1 and C.

For a window size of five, this has the effect that context positions -1 and +1 are included 100% of the time, positions -2 and +2 are included 80% of the time, -3 and +3 are included 60%, -4 and +4 is 40%, and -5 and +5 are only included 20% of the time.

So where does the confusion about this come from? I believe it's from the following illustration, which I have seen used in many tutorials. The illustration is correct, but misleading. The multiple outputs, y_1, y_2, and

y_C can be misinterpreted as the network having "C" different output layers, when in fact it has only one.



Is the illustration wrong? No. First, notice that there are only two unique weight matrices shown here, W and W'. If the network had a different output for each context position, then there would need to be different output weight matrices for each position.

Second, notice the illustration of the outputs--each output contains all empty circles and one black circle--this shows that these represent one-hot vectors. The 'y' vectors in this illustration are simply the training outputs, not the probability distributions that you get when *evaluating* the network.

So what is actually meant by the vectors y_1 through y_C? This illustration is merely conveying that a single training input word (x_k) can have "C" training output words.

A.4. How can the probabilities sum to one if certain words always appear together?

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.

The output of the skip-gram neural network is more like the second of these two interpretations.

A.5. Is the word2vec model really just a single layer neural network, with word vectors as inputs?

Yes and no... First, yes, the one-hot vector in the "input layer" of the neural network is just a mathematical construct. In practice (and in the code) this one-hot vector doesn't appear. Instead, it does look like a single layer, with the word vectors as inputs. However, both the output layer *and the input vectors* are learned through backprop, which is not the norm for a neural network.

B. Bonus #2 - Resources

B.1. Original Papers & Code

Efficient Estimation of Word Representations in Vector Space

Link to paper

This was the first paper, dated September 7th, 2013. It introduces the Continuous Bag of Words (CBOW) and Skip-Gram models.

Distributed Representations of Words and Phrases and their Compositionality

Link to paper

This was a follow-up paper, dated October 16th, 2013. It adds the additional topics (which are also fundamental to word2vec, and present in all word2vec implementations):

1. Subsampling frequent words.
2. Negative Sampling & Hierarchical Softmax
3. Phrase detection

Original C code implementation

Link to code

Their C code implementation also serves as a good reference for understanding the model.

I have published a commented (but otherwise unaltered) version of their code here.

B.2. Understanding the Math

The following are some popular resources for understanding word2vec which also dive into the mathematical formulation.

Stanford CS 224D: Deep Learning for NLP

Francois Chaubard, Rohit Mundra, Richard Socher
https://cs224d.stanford.edu/lecture_notes/notes1.pdf

word2vec Parameter Learning Explained

Xin Rong, June 5th, 2016
https://arxiv.org/pdf/1411.2738.pdf

word2vec Explained: Deriving Mikolov et al.'s Negative-Sampling Word-Embedding Method

Yoav Goldberg and Omer Levy, February 14, 2014
https://arxiv.org/pdf/1402.3722

B.3. Survey of Implementations

- *gensim* - Python library that's very popular for training word2vec models.
- *fastText* - This is also Mikolov's work, which he did after leaving Google for Facebook. It includes some new innovations to word embeddings beyond word2vec. This library is written in C++, only supports Linux and OS X, and seems to be primarily designed to be used as a command-line tool. It does include a Python wrapper, though.
    - *gensim* supports the fastText algorithm as well; see here.
- *word2vec* - The original C library, which only supports Linux. There are two variants of this library worth checking out:
    - dav/word2vec - The original implementation, but modified to be buildable on OS X and Windows, and with other fixes.
    - word2vec commented - The original implementation, but with extensive commenting.
- *TensorFlow* - A steep learning curve if you're not already familiar with the framework, but perhaps it's a good route if you plan to make modifications to the algorithm?
- *Amazon BlazingText* - Also a steep learning curve if you're not familiar with AWS and their SageMaker platform. This implementation is designed for efficiency (to reduce billable time on AWS) and includes GPU support. SageMaker helps with managing the AWS resources required.