1. What's Word2Vec?

The goal of word2vec is to convert each word into a vector representation so that it can be solved by machine learning models.

How do we train a model that can effectively project each word into a vector presentation? What we have is a large number of sentences (millions or billions of words). We will define some rules using the words in the sentences to artificially create input and output of a machine learning model. The end goal is not to train a good model and use the model to make predictions in the future. Instead, what we will keep is only the weight matrix produced by the model training.

In general, there are two different approaches to define the input and output of the machine learning model.

1. Using a center word to predict its context words. This is called Skip-gram model.
2. Using context words to predict the center word. This is called continuous bag of words (CBOW) model.

We will discuss the Skip-gram model here in details.

2. Skip-gram

2.1 Example to show how to define model input and output

Let's say our corpus is just one sentence:

`natural language processing and machine learning is fun and exciting`

We need to define the window size and word embedding vector size for model training. Let's say the window size is 2 and word embedding vector size is 10.

Then our training examples will be

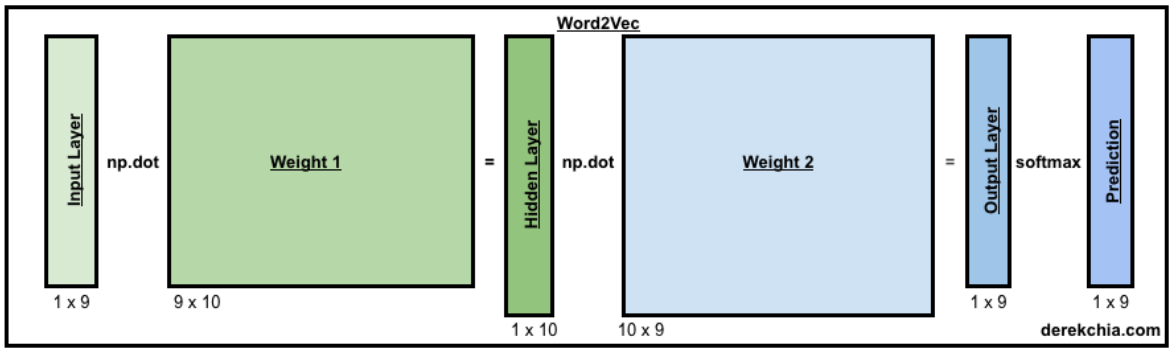
1. (input = [natural], output = [language, processing])
2. (input = [language], output = [natural, processing, and])
3. (input = [processing], output = [natural, language, and, machine])
4. (input = [and], output = [language, processing, machine, learning])
5. (input = [machine], output = [processing, and, learning, is])
6. (input = [learning], output = [and, machine, is, fun])
7. (input = [is], output = [machine, learning, fun, and])
8. (input = [fun], output = [learning, is, and, exciting])
9. (input = [and], output = [is, fun, exciting])
10. (input = [exciting], output = [fun, and])

In order to train them, we first one-hot encode all the words in the corpus. There are 9 unique words in the corpus; therefore the one-hot encoded vector size is 9. Let's say we follow the below sequence to one hot encode the words: `natural language processing and machine learning is fun exciting`. Then for the first and second training examples respectively:

- Input = [1 0 0 0 0 0 0 0 0], output = [0 1 1 0 0 0 0 0 0] (input = [natural], output = [language, processing])

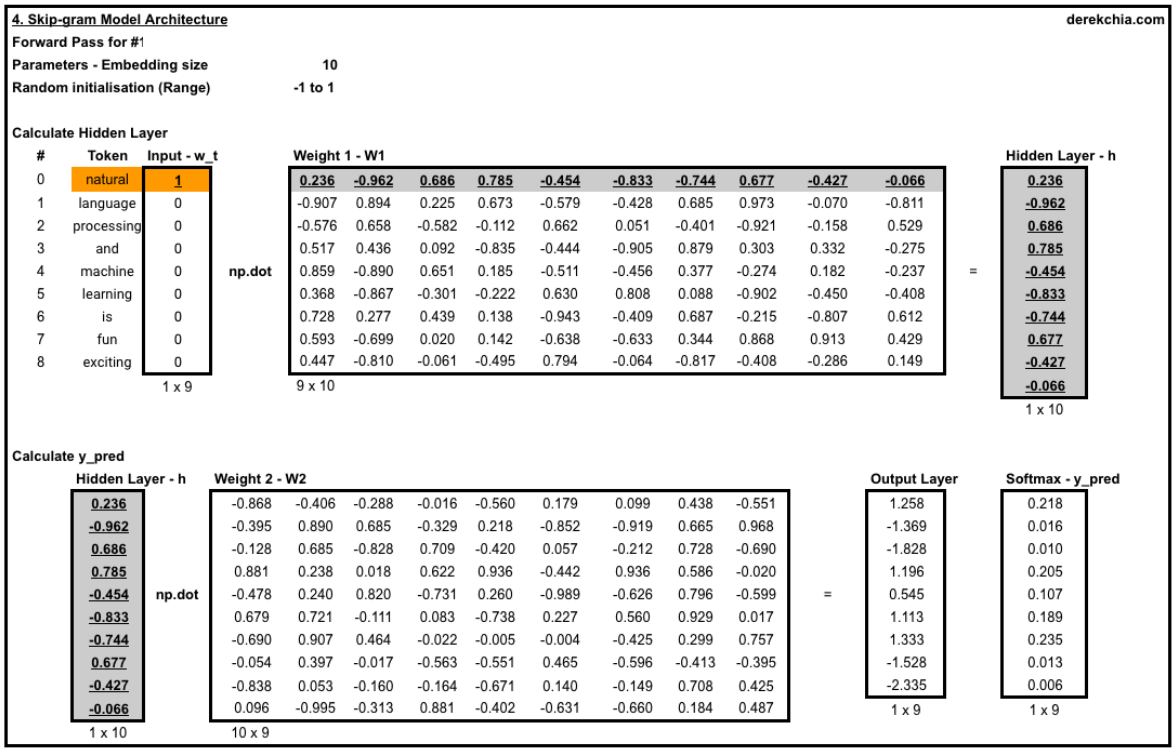
- Input = [0 1 0 0 0 0 0 0 0], output = [1 0 1 1 0 0 0 0 0] (input = [language], output = [natural, processing, and])

As we can see, the input and output vector size of each training example are both 1x9, and the weight matrix has size 9x10. The 10 here is decided by the size of the word embedding vector size.



2.2 Intuition of weight matrix

For training sample1, the input vector is [1 0 0 0 0 0 0 0 0]. Only the first element in the input vector is not zero. Therefore, for this particular training example, only the first row of the weight matrix w1 matters while all other rows in the weight matrix w1 do not affect the matrix multiplication result of the input vector and weight matrix. The weight matrix w1 contains the 9 word vectors of the total 9 unique words in the corpus. However, each training example relates to only one row in the weight matrix for one word. After the model training, row 1 of the weight matrix w1 is the word vector of word 'natural'.



2.3 Softmax output

For the skip-gram model, we use softmax to compute the probability of word j appears in the context of center word i:

$$P(j|i) = \frac{\exp(u_j^T v_i)}{\sum_{w=1}^{W} \exp(u_w^T v_i)}$$

- u_j is the vector of a context word
- u_i is the vector of the center word we use for prediction
- W represents all words in the vocabulary
- w denotes a word in the vocabulary

Note that with standard gradient descent method, you need to compute the probability of all possible words in the vocabulary being a context word to get the value of denominator (normalization factor). It is too computationally expensive. We will use negative sampling to accelerate the process.

2.4 Model training

The model training is just to compute the difference between model prediction $\hat{y}$ and output vector y for each training example. Model prediction $\hat{y}$ is created through a softmax output layer (standard deep learning stuff). We can use cross entropy loss and back propagation for model training (standard deep learning stuff again).

3. Practical challenges of word2vec

The example in section 2 has a corpus of only 9 unique words and a word vector of length 10. However, in reality, there could be 10s of thousands of unique words in the corpus and we may want to train word vectors of length of hundreds. For example, we may have 10,000 unique words in the corpus and want to train a word vector of length 300. Then the weight matrix has size 10,000 x 300, meaning there are 10,000x300 parameters to tune. This will be a nightmare.

The tricks to resolve such issues are:

- Use word pairs and phrases
- Randomly down sample words with higher-frequency
- Negative sampling
- Hierarchical softmax

3.1 Use word pairs and phrases

‘Boston Globe' is totally different from 'Boston' and 'Globe'. 'New York' is totally different from 'New' and 'York'. Therefore, word pairs like these should be treated as one item in the corpus. With this method, the corpus size will be reduced and training will be more effective and more efficient.

Google has created a vocabulary that contains the word pairs and phrases. We can directly look up into the vocabulary to decide whether the words should be bundled or not.

3.2 Randomly down sample words with higher-frequency

Words like 'the', 'that', 'this' occur very frequently in our training data. For a sentence like 'The quick brown fox jumps over the laze dog', input & output combination ('fox', 'the') does not give us much information about the word 'fox'. Therefore, we want to find out a way to down sample words like 'the', 'that', 'this'.

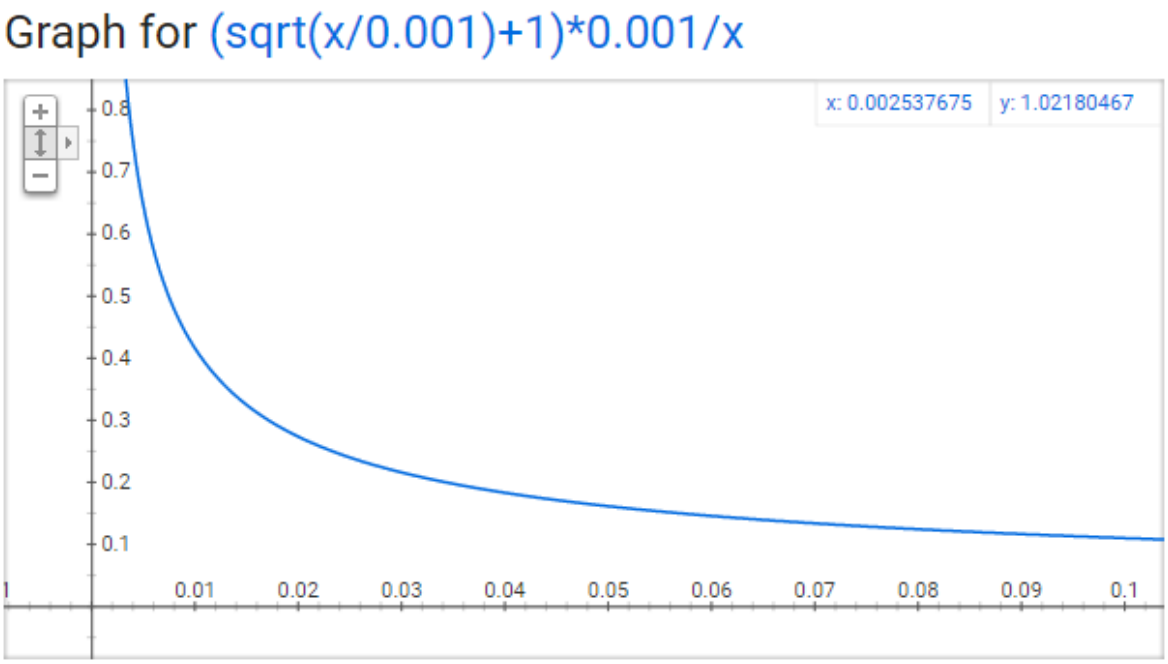
Specifically, for each word we encountered in the training text, we define a rule so that it has a certain probability to be deleted from the text, and the probability of this deletion is related to the frequency of the word appearing in the training text. In the actual implementation, we compute the probability of a word being kept instead:

$$P(w_i) = (\sqrt{\frac{Z(w_i)}{0.001}} + 1) \times \frac{0.001}{Z(w_i)}$$

where

- w_i is a word
- $Z(w_i)$ is the frequency the word appears in the training text. For example, if word 'peanut' appears 1000 times in the training text of size 1,000,000,000, $Z('peanut') = \frac{1000}{1,000,000,000} = 1e^{-6}$
- 0.001 is the threshold for configuring which higher-frequency words are randomly down-sampled. The smaller this value, the lower the probability a word is kept, or the higher the probability a word gets deleted.

some intuitions



In the graph above, x coordinate is $Z(w_i)$, which is the frequency word w_i appears in the training text. y coordinate is the probability a word is being kept $P(w_i)$.

- Usually for a large training text, for each word w_i, $Z(w_i)$ won't be too big.
- As can be seen in the graph, the larger the frequency $Z(w_i)$, the smaller the probability a word is being kept $P(w_i)$.

There are some other interesting findings:

- $P(w_i) = 1$ when $Z(w_i) <= 0.0026$. This means when the frequency of a word w_i appearing in the training text is smaller than 0.0026, it will be 100% kept.
- $P(w_i) = 0.5$ when $Z(w_i) = 0.00746$. This means when the frequency of a word w_i appearing in the training text is 0.00746, there is 50% of chance it will be kept.
- $P(w_i) = 0.033$ when $Z(w_i) =$1.0.

3.3 Negative sampling

The size of the vocabulary is usually humongous, which means the computation cost is huge. In each iteration of gradient descent, you will go through all the combinations of input and output vectors to update the weight matrices. Negative sampling addresses the issue.

When we train our neural network with training example (input=[fox], output=[quick]), both 'fox' and 'quick' are one-hot encoded vectors. If our vocabulary size is 10,000, in the output layer, we expect the neuron node corresponding to the word 'quick' to output 1 and the remaining 9999 neuron nodes to output 0. The words corresponding to the 9999 neuron nodes whose output we expect to be 0 are called **'negative'** words. The word 'quick' is called **'positive'** word.

In negative sampling, we randomly select a small number of negative words instead of using all 9999 words. Usually 5-20 negative words are good for small training text and 2-5 negative words are good for large training text.

For a model with 10,000 unique words in the corpus and word embedding vector of length 300, we have a weight matrix of size 10,000x300. If we select 5 negative words in training, then each iteration we only need to compute a weight matrix of size 6x300. This is only 0.06% of the original weight matrix.

select negative words with unigram distribution

The probability a word to be selected as a negative word is based on the frequency it appears in the training text. The more frequent a word appears in the training text, the more likely it's selected as a negative word during training.

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

where

- $f(w_i)$ is the frequency a word appears in the training text
- 3/4 is purely based on experience
- it makes less frequent words be sampled more often

objective function with negative sampling

Recall that we use softmax function for model output

$$P(j|i) = \frac{\exp(u_j^T v_i)}{\sum_{w=1}^{W}\exp(u_w^T v_i)}$$

If we apply logarithmic function and negative sampling trick

$$J_{neg-sample}(v_j, v_i, U) = -log(\sigma(u_j^T v_i)) - \sum_{k=1}^{K} log(\sigma(-u_k^T v_i))$$

- v_i: vector of center word
- v_j: vector of outside word
- U: unigram distribution
- $\sigma(x) = \frac{1}{1+e^{-x}}$
- K: we take K negative samples using word probabilities
- We maximize probability that real outside word appears and minimize the probability that random words appear around the center word

3.4 Hierarchical softmax

Hierarchical Softmax uses a binary tree where leaves are the words. The probability of a word being the output word is defined as the probability of a random walk from the root to that word's leaf. Computational cost becomes $O(log(|V|))$ instead of $O(|V|)$ where V is the number of unique words in the corpus.