word2vec Implementation Review

Changsoo Kim

Department of Computer Science

University of Illinois at Urbana-Champaign

November 07, 2020

1. Introduction

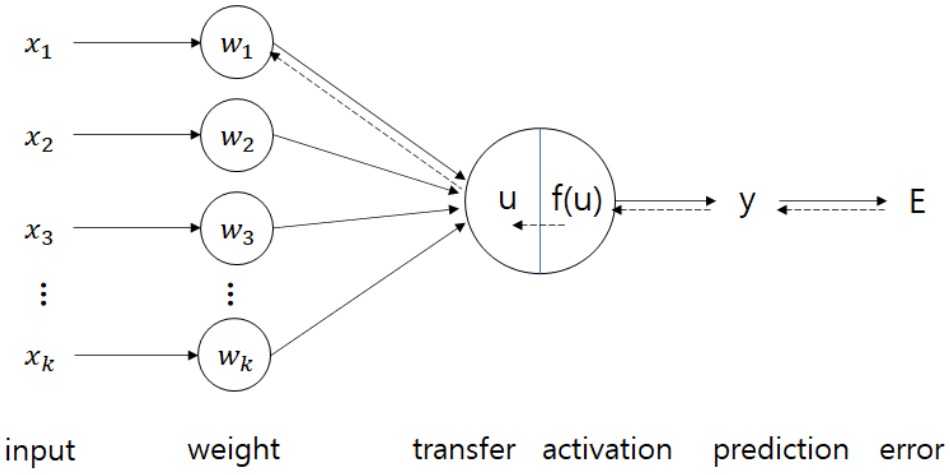
Word 2Vec is a very powerful and basic technique for natural language processing. The algorithm is easy to understand and there are many blogs that explain the model in detail and source codes from scratch[1]. However, if someone is not familiar with the concepts of word embedding, there is an essential prerequisite that needs to be prepared in order to understand the thesis more accurately. So, I will review the model and go through the source code step by step with a graphical interface.

There are already a lot of materials that explain word2vec in detail, so I will only take a brief look. For a computer to process natural language, it must digitize the elements of a sentence. The easiest way to come to mind is one-hot encoding of words. In the case of the statement "Chicago Illinois ", Chicago is expressed as [1 0], and Illinois as [0, 1]. This method takes up more space as there are more words, and it cannot show any association between words. To solve this problem, you can create a model that expresses similar words in a similar vector space by using information that words that occur in the same context are likely to have similar meanings. The training goal of this model is to minimize the difference between the one-hot vector and the result vector of the model. This method saves huge space by storing it in a much lower dimension vector and allows you to calculate relationships between words.

This note checks how the actual matrix element is structured and how to get the differential equation of the cost function through the backpropagation and added a few caveats for Python implementation. Furthermore, I write a python notebook with an interactive GUI to understand the entire flow of the model at a glance.

2. Backpropagation Basics

In machine learning, backpropagation is to find the slope of the Loss function. By finding the slope, the weight can be changed to reduce the loss. Use the chain rule to multiply the partial derivative in the reverse direction.



Let us select only the relevant lines for convenience.

$$x_k \rightarrow w_i \rightarrow u \rightarrow f(u) \rightarrow y \rightarrow E$$

We are interested in how much we need to change w to reduce the total loss amount, E. Therefore, we take the derivative of E with regard to w_i,

$$x_k \rightarrow w_i \rightarrow \partial u \rightarrow \partial f(u) \rightarrow \partial y \rightarrow \partial E$$

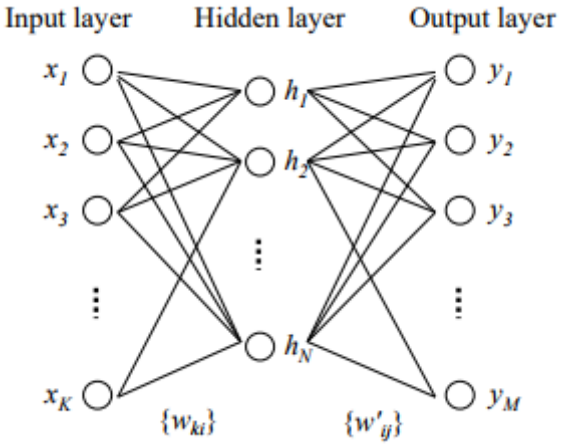
$$\partial w_i \quad \partial u \quad \partial f(u) \quad \partial y$$

$$\frac{\partial E}{\partial y} \cdot \frac{\partial y}{\partial f(u)} \cdot \frac{\partial f(u)}{\partial u} \cdot \frac{\partial u}{\partial w_i} \xleftarrow{\text{backpropagation}} \frac{\partial E}{\partial w_i}$$

Once we found the delta of E, we can move a little to reduce the loss.

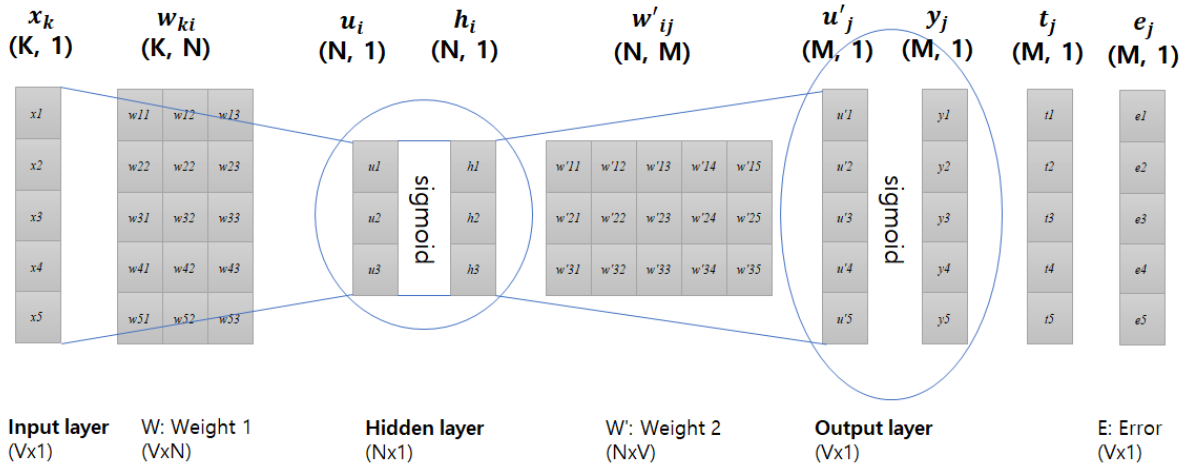
3. Backpropagation with Multi-Layer Network

I found a very detailed word2vec document by Rong[2]. I will just add a little extra explanation here, so I brought the picture from his document.



A multi-layer neural network with one hidden layer

Since there are many variables here, it can be confusing, so I added a picture that expresses all variables at a glance. In this figure, it is assumed that K=5, N=3, and M=5 to get a more specific sense.



A multi-layer neural network implementation example. (K=5, N=3, M=5)

If assume that we use a sigmoid function as an activation function and use a squared sum error

function as a loss function, then we can now derive the following formulas.

$$E = \frac{1}{2}\sum_{j=1}^{M}(t_j - y_j)^2$$

$$y_j = \sigma(u'_j)$$

$$u'_j = h_1 w'_{1j} + h_2 w'_{2j} + \ \ldots + h_i w'_{ij} = \sum_{i=1}^{N} h_i w'_{ij}$$

$$h_i = \sigma(u_i)$$

$$u_i = x_1 w_{1j} + x_2 w_{2j} + \ \ldots + x_k w_{ki} = \sum_{k=1}^{K} x_i w_{ki}$$

Let us do partial differentiation based on the previous formulas for later use.

$$\frac{\partial E}{\partial y_j} = (y_j - t_j)$$

$$\frac{\partial y_j}{\partial u'_j} = y_j(y_j - t_j)$$

$$\frac{\partial u'_j}{\partial w'_{ij}} = h_i$$

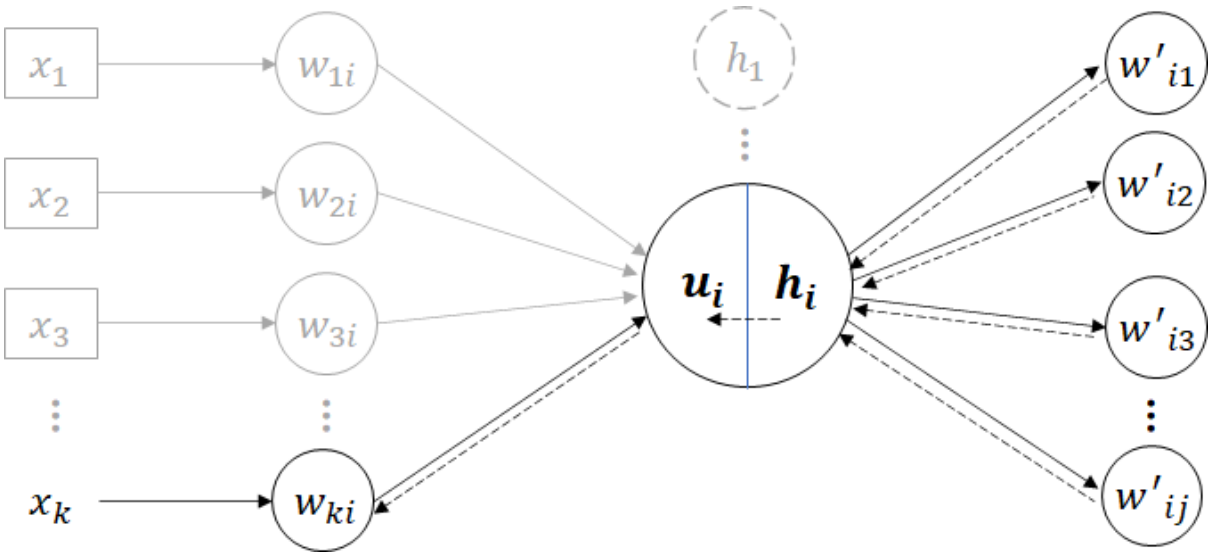
$$\frac{\partial u'_j}{\partial h_i} = w'_{ij}$$

$$\frac{\partial h_i}{\partial u_i} = h_i(1 - h_i)$$

$$\frac{\partial u_i}{\partial w_{ki}} = x_k$$

Remember that in the formulas 1 and 2, all things other than hi in partial derivative can be thought of as constants. Now let us find the slope of the two weights, w_{ki} and w'_{ij} to minimize the error. Here too, I have the same notation of Rong. Except for the fact that all output layers have to be summed for the backpropagation of the hidden layer, there is no difficult part

with a simple formula substitution.



The output of hidden layer is related to all weights in the output layer.

$$\frac{\partial E}{\partial h_i} = \sum_{j=1}^{M} \frac{\partial w'_{ij}}{\partial h_i} \cdot \frac{\partial E}{\partial w'_{ij}}$$

Here is the overall backpropagation flow.

$$x_k \rightarrow w_{ki} \rightarrow u_i \rightarrow h_i \leftrightarrow w'_{ij} \rightarrow u'_j \rightarrow y_j \rightarrow E$$

$$\frac{\partial u_i}{\partial w_{ki}} \quad \frac{\partial h_i}{\partial u_i} \quad \frac{\partial w'_{ij}}{\partial h_i} \quad \frac{\partial u'_j}{\partial w'_{ij}} \quad \frac{\partial y_j}{\partial u'_j} \quad \frac{\partial E}{\partial y_j}$$

x_k $\qquad$ h_i $\qquad$ $\frac{\partial E}{\partial u'_j} = (y_j - t_j) \cdot y_j(1 - y_j) = EI'_j$

$$\frac{\partial u'_j}{\partial h_i} = w'_{ij}$$

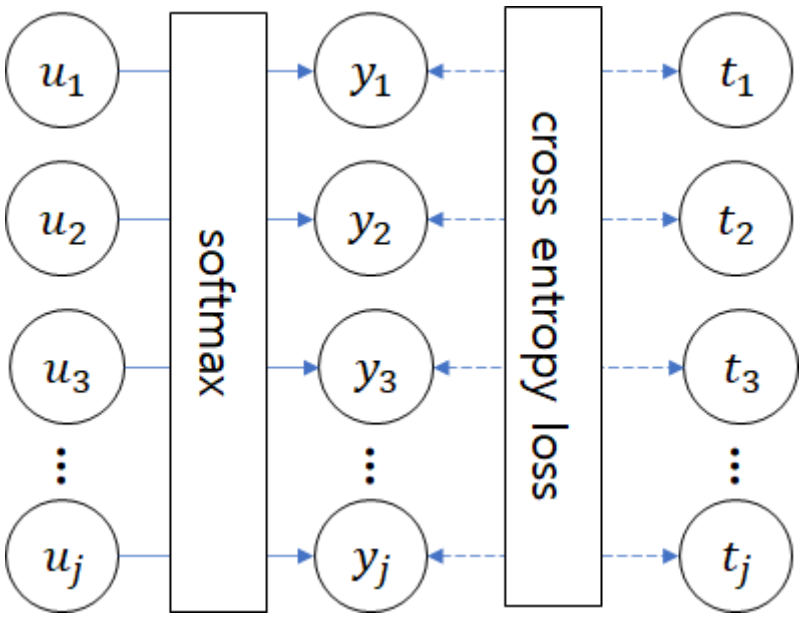
$$\frac{\partial E}{\partial h_i} = \sum_{j=1}^{M} EI'_j \cdot w'_{ij}$$

$$\frac{\partial E}{\partial u_i} = \sum_{j=1}^{M} EI'_j \cdot w'_{ij} \cdot h_i(1 - h_i) = EI_i$$

$$\frac{\partial E}{\partial w_{ki}} = EI_i \cdot x_k$$

4. Softmax + Cross-entropy loss Backpropagation

Technically, there is no term as such Softmax loss[3]. However, people use "softmax" to refer to "softmax + cross-entropy loss function". Softmax is a normalization function that makes the sum of outputs 1. The cross-entropy function is a loss function indicating the degree of the difference between the predicted value and the actual value. When the softmax function and the cross-entropy loss function are used together, the following final formula can be obtained.



An output layer with softmax classifier and cross-entropy loss function

$$\frac{\partial(total\ loss)}{\partial u_j} = \frac{\partial E}{\partial u_j} = y_j - t_j$$

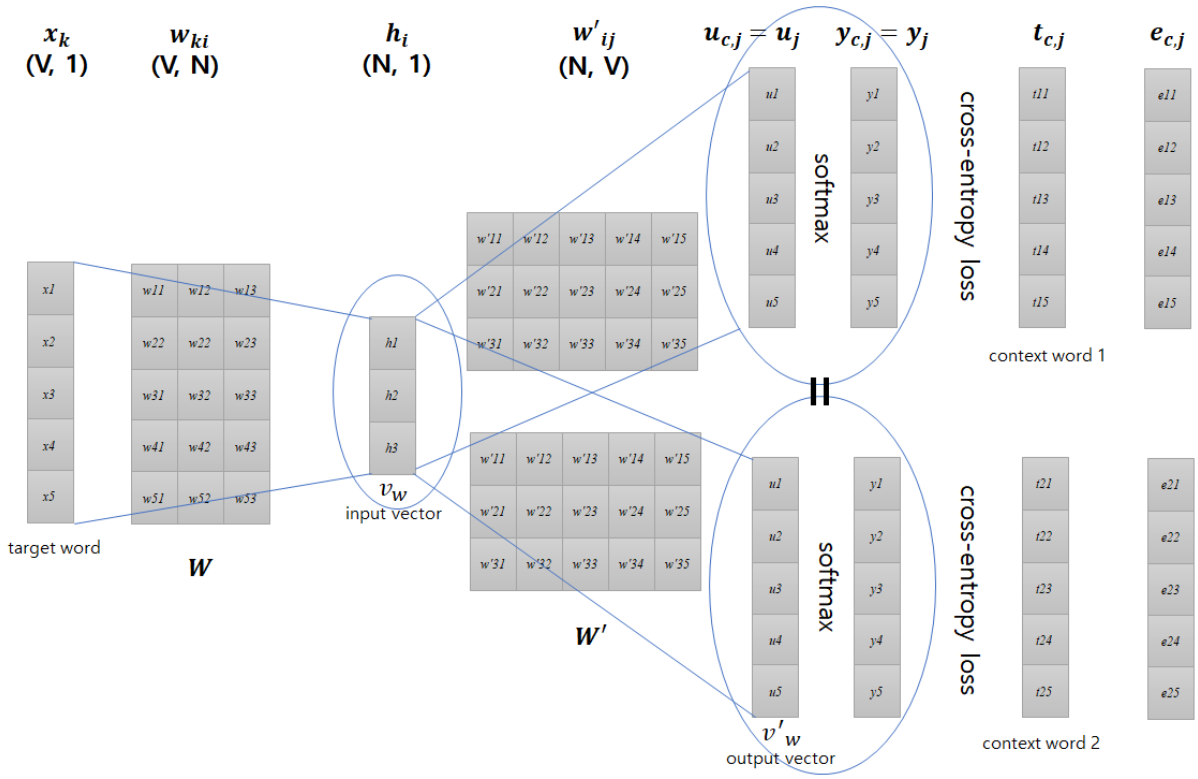
The detailed differential formula is complicated to deal with here, so please refer to the following site. [Here[4]]

5. Numpy caveat

In this note, we will use numpy to implement the Skip-gram model. There are a few caveats while using numpy, so take a look in advance.

- In numpy, the shape of 1D array is represented as (N,).
- The transpose of a 1D array is still a 1D array. (Nothing happens.)
- If a is an (M, N) array and b is a 1-D array and you want to do "dot product" operation, the shape of b should be (N,) and the result is (M,) 1D array.
- Numpy outer function (numpy.outer) compute the outer product of two vectors, (M,) and (N,) 1D arrays. The result is (M, N) ndarray.

6. Skip-gram model



A skip-gram model implementation example. (V=5, N=3, window_size = 1)

With the exception of a few formulas below, the overall process is the same as discussed earlier.

$$u_{c,j} = h_1 w'_{1j} + h_2 w'_{2j} + \ \ldots + h_i w'_{ij} = \sum_{i=1}^{N} h_i w'_{ij}$$

$$h_i = x_1 w_{1j} + x_2 w_{2j} + \ \ldots + x_k w_{ki} \ = \sum_{k=1}^{K} x_i w_{ki}$$

Once again, we can do partial differentiation for later use.

$$\frac{\partial u_{c,j}}{\partial w'_{ij}} = h_i$$

$$\frac{\partial u_{c,j}}{\partial h_i} = \ w'_{ij}$$

$$\frac{\partial h_i}{\partial w_{ki}} \ = x_k$$

Now, we can write

$$x_i \rightarrow \boxed{w_{ki}} \rightarrow h_i \leftrightarrows \boxed{w'_{ij}} \rightarrow u_{c,j} \leftrightarrows (softmax \ \& \ cross\_entropy) \rightarrow E$$

$$\frac{\partial h_i}{\partial w_{ki}} \quad \frac{\partial w'_{ij}}{\partial h_i} \quad \frac{\partial u_{c,j}}{\partial w'_{ij}} \quad \frac{\partial E}{\partial u_{c,j}}$$

$$\leftarrow \quad \leftarrow \quad \leftarrow$$

$$x_k \qquad h_i \qquad \frac{\partial E}{\partial u_{c,j}} = y_{c,j} - t_{c,j} = e_{c,j}$$

$$\leftarrow$$

$$\sum_{c=1}^{C} \frac{\partial E}{\partial u_{c,j}} \ = \sum_{c=1}^{C} e_{c,j} = EI_j$$

$$\leftarrow$$

$$\boxed{\frac{\partial E}{\partial w'_{ij}} = EI_j \cdot h_i}$$

$$\leftarrow$$

$$\frac{\partial u_{c,j}}{\partial h_i} = w'_{ij}$$

$$\leftarrow$$

$$\frac{\partial E}{\partial h_i} = \sum_{j=1}^{V} EI_j \cdot w'_{ij}$$

$$\leftarrow$$

$$\boxed{\frac{\partial E}{\partial w_{ki}} = \sum_{j=1}^{V} EI_j \cdot w'_{ij} \cdot x_k}$$

However, this is an element-based formula, but we want to get the matrix-based formula for an easy implementation. Let us calculate the delta of E with regard to W assuming the first context word C=1. We use $\otimes$ to denote the outer product of two vectors.

$$\mathbf{u} = (u_1, u_2, \ldots, u_m)$$
$$\mathbf{v} = (v_1, v_2, \ldots, v_n)$$

$$\mathbf{u} \otimes \mathbf{v} = \mathbf{A} = \begin{bmatrix} u_1 v_1 & u_1 v_2 & \ldots & u_1 v_n \\ u_2 v_1 & u_2 v_2 & \ldots & u_2 v_n \\ \vdots & \vdots & \ddots & \vdots \\ u_m v_1 & u_m v_2 & \ldots & u_m v_n \end{bmatrix}$$

An outer product of two vectors, denoted $u \otimes v$.

$$\frac{\partial E}{\partial w'_{ij}} = e_{c,j} \cdot h_i$$

$$h = [h_1, h_1, \ldots, h_1]$$

$$e = [e_{1,1}, e_{1,1}, \ldots, e_{1,1}]$$

$$\frac{\partial E}{\partial W'} = \mathrm{h} \otimes e = \begin{bmatrix} h_1 e_{1,1} & \cdots & h_1 e_{1,j} \\ \vdots & \ddots & \vdots \\ h_i e_{1,j} & \cdots & h_i e_{1,j} \end{bmatrix} (N, V)$$

Now, let us calculate the delta of E with regard to W' assuming the first context word C=1.

$$\frac{\partial E}{\partial w_{ki}} = \sum_{j=1}^{V} e_{c,j} \cdot w'_{ij}$$

$$x = [x_1, x_2, \ldots, x_k]\ (V,)$$

$$e = [e_{1,1}, e_{1,2}, \ldots, e_{1,j}]\ (V,) \rightarrow e^T\ (V,)$$

$$W' = \begin{bmatrix} w'_{11} & w'_{12} & \cdots & w'_{1j} \\ \vdots & \ldots & \ddots & \vdots \\ w'_{i1} & w'_{i2} & \cdots & w'_{ij} \end{bmatrix} (N, V),\ e^T = [e_{1,1}, e_{1,2}, \ldots, e_{1,j}]\ (V,)$$

$$W' \cdot e^T = [e_{1,1} w'_{11} + \cdots + e_{1,1} w'_{1j},\ \ldots, e_{1,j} w'_{i1} + \cdots + e_{1,j} w'_{ij}]\ (N,)$$

$$\frac{\partial E}{\partial W} = x \otimes (W' \cdot e^T) = x^T \cdot (W' \cdot e^T) =$$

$$\begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_k \end{bmatrix} [e_{1,1}w'_{11} + \cdots + e_{1,1}w'_{1j},\ \dots, e_{1,j}w'_{i1} + \cdots + e_{1,j}w'_{ij}] =$$

$$\begin{bmatrix} 1 \\ 0 \\ \dots \\ 0 \end{bmatrix} \left[\sum_{j=1}^{V} e_{1,j} \cdot w'_{1j},\ \dots, \sum_{j=1}^{V} e_{1,j} \cdot w'_{ij} \right]$$

In sum, we take the derivative of E with regard to W and W':

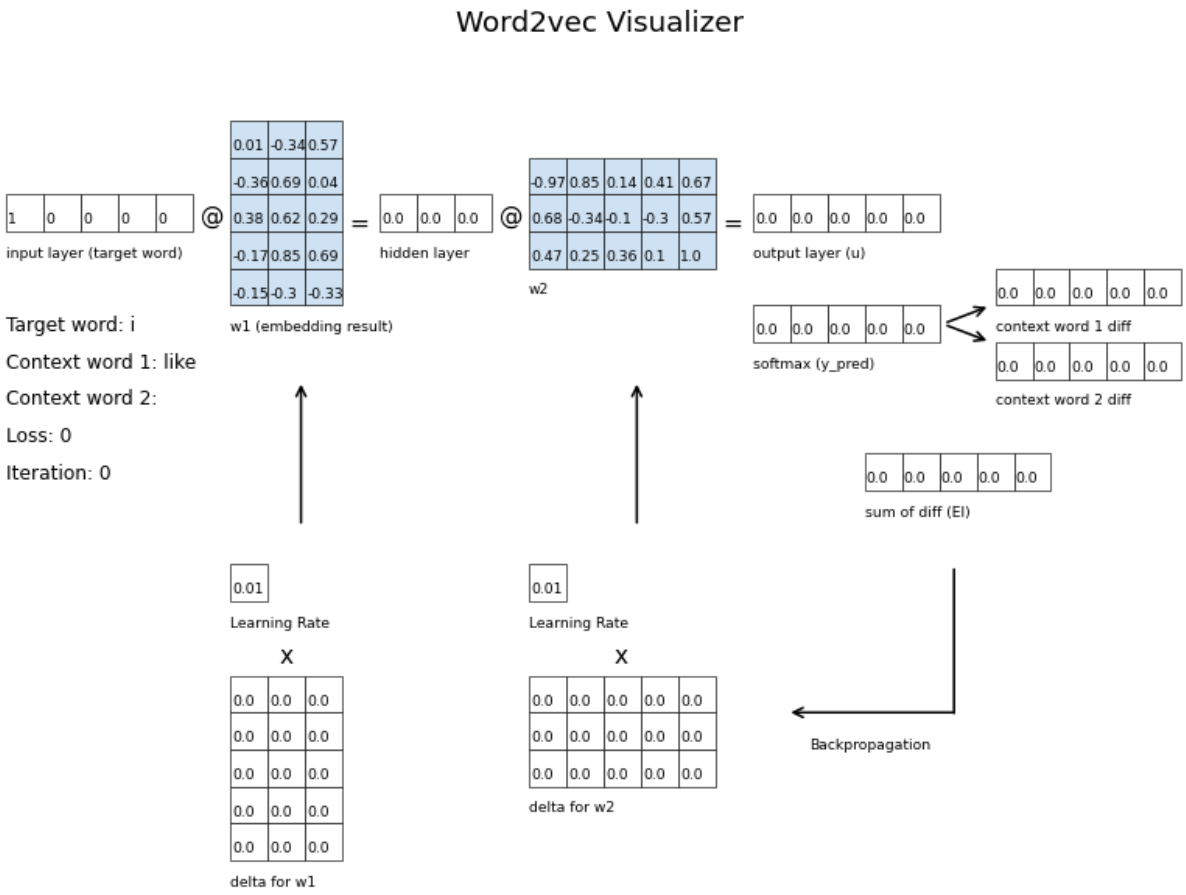
$$\frac{\partial E}{\partial W'} = \mathrm{h} \otimes e$$

$$\frac{\partial E}{\partial W} = x \otimes (W' \cdot e^T)$$

Now we just need to run this formula for each context word for one target word. Until now, we make a formula for one target word. To complete one training iteration, do this for all vocabularies (target words).

7. Training process visualize

So far, we have seen the detailed process of the model. However, the actual python implementation has a slightly different data shape. For example, the input vector is not a two-dimensional matrix of (V, 1), but a one-dimensional matrix of (V,). Because these small changes can cause confusion, I made a program that can easily see the changes of actual variables step by step. You can click the graph to see the next step. Here[5] is a source code.



A screenshot of the word2vec visualizer. (V=5, N=3, window_size=1)

8. Conclusion

So far, you have seen all the very basic things you need to implement word2vec model. As mentioned in the original paper[6], "Efficient Estimation of Word Representations in Vector Space ", word2vec do not have a nonlinear hidden layer unlike NNLM or RNNLM, so the computational overhead is greatly reduced. In addition, techniques such as Hierarchical softmax, Negative sampling, and Subsampling of Frequent words are added to reduce the computational complexity. If we use the intuition discussed here, it will be much easier to understand and implement the boilerplate code of the similar model.

References

[1] An implementation guide to Word2Vec using NumPy and Google Sheets https://towardsdatascience.com/an-implementation-guide-to-word2vec-using-numpy-and-google-sheets-13445eebd281

[2] word2vec Parameter Learning Explained https://arxiv.org/abs/1411.2738

[3] The true meaning of softmax loss https://www.quora.com/Is-the-softmax-loss-the-same-as-the-cross-entropy-loss

[4] Derivative of the cross-entropy loss function for the softmax function https://peterroelants.github.io/posts/cross-entropy-softmax/

[5] word2vec implementation with an interactive GUI https://github.com/tiratano/tech_review/blob/master/word2vec_simulator.ipynb

[6] Efficient Estimation of Word Representations in Vector Space https://arxiv.org/pdf/1301.3781.pdf