Math in a Nutshell #MiaN

CONTINUOUS BAG OF WORDS

Disclaimer
As suggested in the title, this note explains the mathematical fundamentals without considering any optimization techniques such as regularization, negative sampling or hierarchical softmax. They would require modifications in the formulas but the core ideas remain.

This note does not aim to explain what CBOW (or even word embedding) is all about, but instead I do expect readers to be familiar with these concepts. I also assume background knowledge in calculus, linear algebra, machine learning and neural network.

At any point will you have trouble comprehending the concepts, feel free to pause and go through the following materials, which contribute to deepening my understanding. It is totally normal to go back and forth among these papers / blogs.

Reference

(Original paper)
Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

Wang, B., Wang, A., Chen, F., Wang, Y., & Kuo, C. C. J. (2019). Evaluating word embedding models: Methods and experimental results. *APSIPA transactions on signal and information processing*, *8*.

Rong, X. (2014). word2vec parameter learning explained. *arXiv preprint arXiv:1411.2738*.

Lecture Notes 1 – Stanford CS 224D: Deep Learning for NLP
https://cs224d.stanford.edu/lecture_notes/LectureNotes1.pdf

Data
A corpus of text.
If you have a dataset storing multiple documents, concatenate them into a giant corpus, but don't mess up the order as the model takes it into account.

All the words in the corpus form a vocabulary.

Notations
Vocabulary size: N
Embedding (or hidden layer) size: n
Window size: m
Learning rate: η
A word at jth position in vocabulary: w_j
a_k is the kth element of any vector $\vec{a}$ size (N x 1) with k: 1 → N
b_i is the ith element of any vector $\vec{b}$ size (n x 1) with i: 1 → n

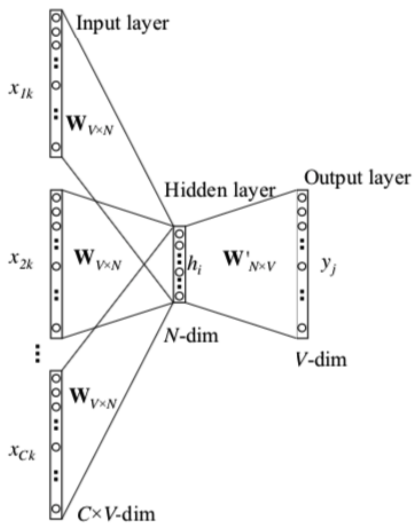
Goal
To predict a center word w_j given its context words

$$(w_{j-m}, \quad \ldots, \quad w_{j-2}, \quad w_{j-1}, \quad w_{j+1}, \quad w_{j+2}, \quad \ldots, \quad w_{j+m})$$

For a given center word w_j, the number of its context words (both sides) is c_j

Throughout the optimization process, we also aim to get the vector representations of each word in vocabulary.

A stochastic network of CBOW model for a single center word w_j



Inputs / Input layer
Each word w_j is represented by a one-hot vector size (N x 1) with jth unit = 1. The word has c_j context words, each of which is also represented by a one-hot vector x_{j-m} size (N x 1) with its corresponding index = 1***

Weights
- Input weights V size (n x N)
- Output weights U size (N x n)

Matrix U contains numeric representations of all N words in vocabulary.

Outputs / Output layer
One-hot vector Y representing actual center word j to be predicted size (N x 1) with jth unit = 1.

We also denote $\hat{Y}$ as the predicted (probability) center word vector. Values of its elements are between 0 and 1. The predicted word is the one at which the probability is the highest.

*****Input vectors**
As the center word j has multiple context words, I present two ways to look at how the input vector is derived – one from the original paper and the other from my implementation.

Original paper version
1. Get embedded word vectors for each context word, size (N x 1) each

$$v_{j-m} = Vx_{j-m}$$
$$\vdots$$
$$v_{j-1} = Vx_{j-1}$$
$$v_{j+1} = Vx_{j+1}$$
$$\vdots$$
$$v_{j+m} = Vx_{j+m}$$

2. Average these vectors to get hidden layer vector h size (n x 1)

$$h = \frac{v_{j-m} + \cdots + v_{j-1} + v_{j+1} + \cdots + v_{j+m}}{c_j}$$

My version

3. Average context vectors to get X size (N x 1)

$$X = \frac{x_{j-m} + \cdots + x_{j-1} + x_{j+1} + \cdots + x_{j+m}}{c_j}$$

4. Pass this vector X through the network as in single context model

$$h = V.X$$

In general, the model goes as follow

$$\boxed{\boldsymbol{V}^{\boldsymbol{n\times N}}.\ \boldsymbol{X}^{\boldsymbol{N\times 1}} = \boldsymbol{h}^{\boldsymbol{n\times 1}} \rightarrow \boldsymbol{U}^{\boldsymbol{N\times n}}.\ \boldsymbol{h}^{\boldsymbol{n\times 1}} = \boldsymbol{z}^{\boldsymbol{N\times 1}} \rightarrow \boldsymbol{softmax}(\boldsymbol{z})^{\boldsymbol{N\times 1}} = \hat{\boldsymbol{Y}}^{\boldsymbol{N\times 1}}}$$

- $\boldsymbol{V}^{\boldsymbol{n\times N}}.\ \boldsymbol{X}^{\boldsymbol{N\times 1}} = \boldsymbol{h}^{\boldsymbol{n\times 1}}$

$$h = \begin{pmatrix} h_1 \\ h_2 \\ \vdots \\ h_n \end{pmatrix} = \begin{pmatrix} v_{11} & \dots & u_{1N} \\ \vdots & \vdots & \vdots \\ v_{n1} & \dots & u_{nN} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_N \end{pmatrix}$$

$$h_i = v_{i1}x_1 + u_{i2}x_2 + \cdots + u_{iN}x_n \ (1)$$

- $\boldsymbol{U}^{\boldsymbol{N\times n}}.\ \boldsymbol{h}^{\boldsymbol{n\times 1}} = \boldsymbol{z}^{\boldsymbol{N\times 1}}$

$$z = \begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_N \end{pmatrix} = \begin{pmatrix} u_{11} & \dots & u_{1n} \\ \vdots & \vdots & \vdots \\ u_{N1} & \dots & u_{Nn} \end{pmatrix} \begin{pmatrix} h_1 \\ \vdots \\ h_n \end{pmatrix}$$

$$z_k = u_{k1}h_1 + u_{k2}h_2 + \cdots + u_{kn}h_n \ (2)$$

- $\boldsymbol{softmax}(\boldsymbol{z})$

$$z = \begin{pmatrix} z_1 \\ z_2 \\ \vdots \\ z_N \end{pmatrix} \rightarrow \hat{Y}^{N\times 1} = softmax(z) = \begin{pmatrix} softmax(z_1) \\ softmax(z_2) \\ \vdots \\ softmax(z_N) \end{pmatrix} = \begin{pmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_N \end{pmatrix}$$

With z_k, y_k is the kth element of vector z and y

$$\hat{y}_k = softmax(z_k) = \frac{\exp(z_k)}{\sum_{a=1}^{N} \exp(z_a)} = \frac{\exp(z_k)}{\exp(z_1) + \dots + \exp(z_a) + \dots + \exp(z_N)}$$

Softmax is used to make sure $\hat{y}_k$ values represent probabilities (between 0 and 1).

Loss function / Cross entropy

$$\boldsymbol{L = -Y log(\hat{Y})}$$

Recall that Y is a one-hot vector with only j_{th} unit = 1. Hence,

$$\boldsymbol{L = -y_j log(\hat{y}_j) = -log\frac{\exp(z_j)}{\sum_{k=1}^{N}\exp(z_k)} = log\sum_{k=1}^{N}\exp(z_k) - z_j} \quad \mathbf{(3)}$$

Note that L is a scalar.

Optimization

The objective is to find the optimal set of weights that minimize the loss function, which basically measures the discrepancy between the actual and predicted output - the error. The lower the cost, the closer the predicted output approximates the actual.

Backward Propagation

Update weights for hidden → output weights U

For each component u_{ki} of matrix U, we update

$$u_{ki}^{(new)} = u_{ki}^{(old)} - \eta\frac{\partial L}{\partial u_{ki}}$$

(2) shows that changes in u_{ki} affect L through z_k only.
By the Chain Rule,

$$\frac{\partial L}{\partial u_{ji}} = \frac{\partial L}{\partial z_k}.\frac{\partial z_k}{\partial u_{ji}}$$

We have

$$\frac{\partial L}{\partial z_k} = \frac{\exp(z_k)}{\sum_{a=1}^{N}\exp(z_a)} - t_k = \hat{y}_k - t_k$$

Since L also depends on z_j (jth component of vector z)

$$t_k = \frac{\partial z_j}{\partial z_k} = \begin{cases} 1 \; if \; j = k \\ 0 \; if \; j \neq k \end{cases}$$

which is equivalent to

$$\frac{\partial L}{\partial z_k} = \hat{y}_k - y_k = e_k$$

with y_k the kth component of actual ouput vector Y since Y is one-hot.
Also, through (2) we know that $\frac{\partial z_k}{\partial u_{ji}} = h_i$. So

$$\frac{\partial L}{\partial u_{ji}} = e_k.h_i$$

Update weights for input → hidden weights V

For each component v_{ik} of matrix U, we update

$$v_{ik}^{(new)} = v_{ik}^{(old)} - \eta \frac{\partial L}{\partial v_{ik}}$$

Similarly, (1) shows that changes in v_{ik} affects h_i only. So,

$$\frac{\partial L}{\partial v_{ik}} = \frac{\partial L}{\partial h_i} . \frac{\partial h_i}{\partial u_{ik}}$$

It can be easily derived that

$$\frac{\partial h_i}{\partial u_{ik}} = x_k$$

From (2), we know that changes in h_i affects L by affecting all components z_k of z. Thus changes in L with respect to h_i is the summation of changes in all $z_k's$ with k from 1 to N.

$$\frac{\partial L}{\partial h_i} = \sum_{k=1}^{N} \frac{\partial L}{\partial z_k} . \frac{\partial z_k}{\partial h_i} = \sum_{k=1}^{N} e_k . u_{ki}$$

After all, we have

$$\frac{\partial L}{\partial v_{ik}} = \left(\sum_{k=1}^{N} e_k . u_{ki} \right) . x_k$$

As for implementation, we wish to update all the weights at once through matrix manipulation. The updates for each element of U and V can be translated to

$$\boldsymbol{U}^{(\boldsymbol{new})} = \boldsymbol{U}^{(\boldsymbol{old})} - \boldsymbol{\eta}\, \boldsymbol{e}^{\,\boldsymbol{N \times 1}} . (\boldsymbol{h_T})^{\,\boldsymbol{1 \times n}}$$

with $e = \begin{pmatrix} e_1 \\ e_2 \\ \vdots \\ e \end{pmatrix}$ and h_T is the transpose of h

and

$$\boldsymbol{V}^{(\boldsymbol{new})} = \boldsymbol{V}^{(\boldsymbol{old})} - \boldsymbol{\eta}\, (\boldsymbol{U_T})^{\boldsymbol{n \times N}} . \boldsymbol{e}^{\,\boldsymbol{N \times 1}} . (\boldsymbol{X_T})^{\,\boldsymbol{1 \times N}}$$

with U_T is the transpose of U and X_T is the transpose of X