Word2Vec – Numerical Optimization Approach

Word2vec is one of the most crucial foundation for modern NLP. It is based on the following two papers:

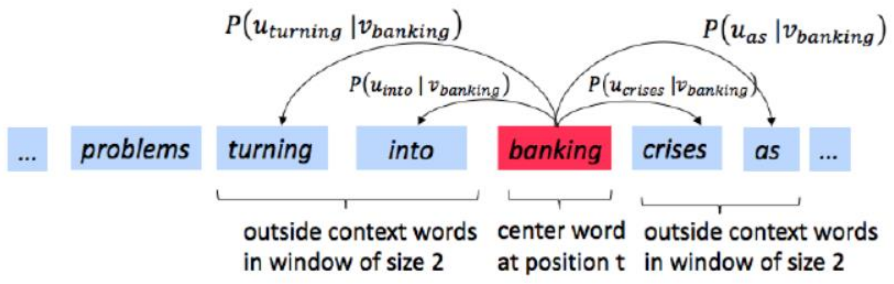
- The original word2vec paper - *Thomas Mikolov et al, 2013, Efficient Estimation of Word Representations in Vector Space*
- The negative sampling paper - *Thomas Mikolov et al, 2013, Distributed Representations of Words and Phrases and their Compositionality*

As a common practice, word2vec is usually implemented through Stochastic Gradient Descent (SGD) with neural networks, and that is the approach proposed in the original papers. Almost one decade after the publication of these original papers, the common practice for word2vec is very similar to the original approach, but will probably be based on one of the deep learning frameworks, such as Tensorflow or Pytorch, which did not exist yet at the time of the publication of the papers.

In this short writing, we would like to present an alternative approach for word2vec, and that is the conventional approach of numerical optimization, which can be implemented with numerical computation packages, such as numpy. We show how to derive the gradients and use them to implement word2vec. Readers who are comfortable with matrix calculus will find this writing helpful for understanding the word2vec algorithm at a deeper level. Nevertheless, the forward propagation of neural network is just a way to calculate the loss function, and the backpropagation of neural networks is just calculating the gradients (derivatives) of the loss function with respect to the parameters through the chain rules of calculus.

The key insight behind word2vec is that "a word is known by the company it keeps". Suppose we have a "center" word *c* and a contextual window surrounding *c*. We shall refer to words that lie in this contextual window as "outside words". For example, in the following figure, the center word *c* is "banking", and suppose the context window size is 2, the outside words are "turning", "into", "crises", and "as".

The goal of the skip-gram word2vec algorithm is to learn the probability distribution P(O|C) as accurately as possible. Given a specific word *o* and a specific word *c*, we want to calculate P(O = *o*|C = *c*), which is the probability that word *o* is an "outside" word for *c*, i.e., the probability that *o* falls within the contextual window of *c*.


$P(u_{turning} | v_{banking})$
$P(u_{as} | v_{banking})$
$P(u_{into} | v_{banking})$
$P(u_{crises} | v_{banking})$
...
problems
turning
into
banking
crises
as
...
outside context words in window of size 2
center word at position t
outside context words in window of size 2

In word2vec, the conditional probability distribution is given by taking vector dot-products and applying the softmax function:

$$P(O = o|C = c) = \frac{\exp(\boldsymbol{u}_o^T \boldsymbol{v}_c)}{\sum_{w \in vocab} \exp(\boldsymbol{u}_w^T \boldsymbol{v}_c)}$$

Notations:

- *d* is the dimension of each word vector, the original word2vec paper has tested *d* for various values, one of them is 300, and that has become the de facto standard default value for the dimension of word embedding since then
- *N* is the size of the vocabulary, the notation for this number is usually |V|, but since we are using $\boldsymbol{V}$ to represent a matrix in the following, so we have chosen to use *N* to avoid the confusion from overloading notations.
- $\boldsymbol{u}_o$ is the "outside" word vector representing outside word *o*, $\boldsymbol{u}_o \in \mathbb{R}^d$
- $\boldsymbol{v}_c$ is the "center" word vector representing center word *c*, $\boldsymbol{v}_c \in \mathbb{R}^d$
- $\boldsymbol{U}$ is a matrix with its columns as all of the "outside" vectors $\boldsymbol{u}_w \in \mathbb{R}^d$
- $\boldsymbol{V}$ is a matrix with its columns as all of the "center" vectors $\boldsymbol{v}_w \in \mathbb{R}^d$
- Both $\boldsymbol{U}$ and $\boldsymbol{V}$ contain a vector for every word $w \in vocab$, $\boldsymbol{U}, \boldsymbol{V} \in \mathbb{R}^{d \times N}$
- $\boldsymbol{y}, \hat{\boldsymbol{y}}$ are vectors with length equal to size of the vocabulary. The i^{th} entry in these vectors indicates the conditional probability of the i^{th} word being an "outside word" for the given "center" word *c*. The true empirical distribution $\boldsymbol{y}$ is a one-hot vector with a 1 for the true outside word *o*, and 0 everywhere else. The predicted distribution $\hat{\boldsymbol{y}}$ is the probability distribution P(O|C = *c*) given by the above-mentioned softmax function, $\boldsymbol{y}, \hat{\boldsymbol{y}} \in \mathbb{R}^N$
- The terms "derivative" and "gradient" are used interchangeably in this writing
- Suppose $\boldsymbol{U}$ is a matrix or vector and *L* is scalar function with $\boldsymbol{U}$ as its parameters (e.g. *L* could be the loss function), the derivative $\frac{\partial L}{\partial \boldsymbol{U}}$ is an element-wise operation, and the outcome is a matrix or vector of the same dimension/shape as $\boldsymbol{U}$
- The following well-known formulae for the sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$ will be used :

  $$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x)), \qquad \sigma(-x) = 1 - \sigma(x)$$

  The trivial proof of these formulae is omitted from here.

Naïve softmax loss function

For a given pair of center word *c* and outside word *o*, the naïve softmax loss function is given by:

$$L_{nv}(\boldsymbol{v}_c, o, \boldsymbol{U}) = -\log P(O = o|C = c) = -\boldsymbol{u}_o^T \boldsymbol{v}_c + \log\left(\sum_{w \in vocab} \exp(\boldsymbol{u}_w^T \boldsymbol{v}_c)\right)$$

We can view this loss as the cross-entropy between the true distribution $\boldsymbol{y}$ and the predicted distribution $\hat{\boldsymbol{y}}$. This is because the "true empirical distribution $\boldsymbol{y}$ is a one-hot vector with a 1 for the true outside word o, and 0 everywhere else". So the cross entropy $-\sum_{i=1}^{N} y_i \log \hat{y}_i$ has only one non-zero term that is corresponding to the true outside word o, $y_o = 1$, therefore the cross entropy is equal to $-\log \hat{y}_o$, which is in turn equal to:

$$-\log P(O = o|C = c) = L_{nv}(\boldsymbol{v}_c, o, \boldsymbol{U})$$

Negative sampling loss function

Assume that K negative samples (words) are drawn from the vocabulary. For simplicity of notation we shall refer to them as w_1, w_2, ..., w_K, and their outside vectors as $\boldsymbol{u}_1$, $\boldsymbol{u}_2$, ..., $\boldsymbol{u}_K$

Note:

- The K words might not be distinct, i.e., it might be true that $w_1 = w_{2,}$ for $i \neq j$
- $o \notin \{w_1, w_2, \ldots, w_K\}$, just to make sure to dump and re-draw a sampled word if it happens to be the same as o

For a given pair of center word c and outside word o, the negative sampling loss function is:

$$L_{ng}(\boldsymbol{v}_c, o, \boldsymbol{U}) = -\log\left(\sigma(\boldsymbol{u}_o^T \boldsymbol{v}_c)\right) - \sum_{k=1}^{K} \log\left(\sigma(-\boldsymbol{u}_k^T \boldsymbol{v}_c)\right)$$

Derivation of gradients for skip-gram algorithm:

In the following, we will derive the gradients of the loss functions $L_{nv}(\boldsymbol{v}_c, o, \boldsymbol{U})$ and $L_{ng}(\boldsymbol{v}_c, o, \boldsymbol{U})$ with respect to $\boldsymbol{U}$ and $\boldsymbol{V}$, based on the chain ruled of calculus. We will write the resulted gradients (derivative) into vectorized format, so to leverage the matrix and vector operations provided by the **numpy** package. Vectorization is crucial for speeding up the algorithm, it is orders of magnitude faster than an implementation with nested loops at the Python language level. That is a common practice for numerical optimization problems such as gradient descent to boost performance. We break down the derivation into the following steps:

1. Compute the partial derivative of $L_{nv}(\boldsymbol{v}_c, o, \boldsymbol{U})$ with respect to $\boldsymbol{v}_c$:

$$\frac{\partial L_{nv}}{\partial \boldsymbol{v}_c} = -\boldsymbol{u}_o + \frac{\sum_{w \in vocab} \exp(\boldsymbol{u}_w^T \boldsymbol{v}_c)\, \boldsymbol{u}_w}{\sum_{w \in vocab} \exp(\boldsymbol{u}_w^T \boldsymbol{v}_c)}$$

   In the above formula, the second term is actually a weighted average of all of the $\boldsymbol{u}_w$, the number of weights is the size of the vocabulary, for a certain word w, the weight corresponding to each word w is $\frac{\exp(\boldsymbol{u}_w^T \boldsymbol{v}_c)}{\sum_w \exp(\boldsymbol{u}_w^T \boldsymbol{v}_c)}$, all of these forms a vector, and it is actually the predicted distribution $\hat{\boldsymbol{y}}$ mentioned above

   Therefore we can rewrite $\frac{\partial L_{nv}}{\partial \boldsymbol{v}_c} = -\boldsymbol{u}_o + \boldsymbol{U}\,\hat{\boldsymbol{y}} = \boldsymbol{U}\,(\hat{\boldsymbol{y}} - \boldsymbol{y})$, which is a vector in the $\mathbb{R}^d$, and can be implemented with numpy matrix multiplication very efficiently.

Where $\boldsymbol{y}$ is defined above as the true distribution, which is a one-hot vector with 1 for the true outside word *o*, and 0 otherwise. $\boldsymbol{U}$ is defined as the matrix to hold all the "outside" vectors $\boldsymbol{u}_w$'s

$\boldsymbol{U} \in \mathbb{R}^{d\times N}, \hat{\boldsymbol{y}} - \boldsymbol{y} \in \mathbb{R}^N$, therefore $\boldsymbol{U}\,(\hat{\boldsymbol{y}} - \boldsymbol{y}) \in \mathbb{R}^d$, the same dimension as $\boldsymbol{v_c}$

2. Compute the partial derivatives of naive-softmax version $L_{nv}(\boldsymbol{v}_c, o, \boldsymbol{U})$ with respect to each of the "outside" word vectors, $\boldsymbol{u}_w's$.

When *w* = *o*

$$\frac{\partial L_{nv}}{\partial \boldsymbol{u}_w} = -\boldsymbol{v}_c + \frac{\exp(\boldsymbol{u}_w^T\boldsymbol{v}_c)\,\boldsymbol{v}_c}{\sum_w \exp(\boldsymbol{u}_w^T\boldsymbol{v}_c)} \quad = (\hat{y}_w - 1)\boldsymbol{v}_c$$

When $w \neq o$

$$\frac{\partial L_{nv}}{\partial \boldsymbol{u}_w} = \frac{\exp(\boldsymbol{u}_w^T\boldsymbol{v}_c)\,\boldsymbol{v}_c}{\sum_w \exp(\boldsymbol{u}_w^T\boldsymbol{v}_c)} = \hat{y}_w\boldsymbol{v}_c$$

We can combine the two cases into one formula:

$$\frac{\partial L_{nv}}{\partial \boldsymbol{u}_w} = (\hat{y}_w - y_w)\boldsymbol{v}_c,$$

Where $y_w = \begin{cases} 1, & w = o \\ 0, & w \neq o \end{cases}$, is the one-hot indicator about whether *w* is the true outside word *o*

With that, we can write the partial derivatives of naive-softmax version $L_{nv}(\boldsymbol{v}_c, o, \boldsymbol{U})$ with respect to $\boldsymbol{U}$ as the following:

$$\frac{\partial L_{nv}}{\partial \boldsymbol{U}} = \boldsymbol{v}_c(\hat{\boldsymbol{y}} - \boldsymbol{y})^T = \boldsymbol{v}_c \otimes (\hat{\boldsymbol{y}} - \boldsymbol{y})$$

$\boldsymbol{v}_c$ is column vector of $\mathbb{R}^d$, $(\hat{\boldsymbol{y}} - \boldsymbol{y})^T$ is a row vector of $\mathbb{R}^{1\times N}$, $\boldsymbol{v}_c(\hat{\boldsymbol{y}} - \boldsymbol{y})^T$ also can be denoted as $\boldsymbol{v}_c \otimes (\hat{\boldsymbol{y}} - \boldsymbol{y})$, the outer product of the two vectors which result in a matrix of $\mathbb{R}^{d\times N}$, that is expected since $\frac{\partial L_{nv}}{\partial \boldsymbol{U}}$ should have the same shape as $\boldsymbol{U} \in \mathbb{R}^{d\times N}$

We can use numpy.outer to implement this outer product very efficiently

3. Compute the partial derivative of $L_{ng}(\boldsymbol{v}_c, o, \boldsymbol{U})$ with respect to $\boldsymbol{v}_c$
Assume the *K* negative sampled words $w_1, w_2, ..., w_K$ are distinct

$$\frac{\partial L_{ng}}{\partial \boldsymbol{v}_c} = -\frac{\sigma(\boldsymbol{u}_o^T\boldsymbol{v}_c)\big(1 - \sigma(\boldsymbol{u}_o^T\boldsymbol{v}_c)\big)}{\sigma(\boldsymbol{u}_o^T\boldsymbol{v}_c)}\boldsymbol{u}_o + \sum_{k=1}^{K}\frac{\sigma(-\boldsymbol{u}_k^T\boldsymbol{v}_c)\big(1 - \sigma(-\boldsymbol{u}_k^T\boldsymbol{v}_c)\big)}{\sigma(-\boldsymbol{u}_k^T\boldsymbol{v}_c)}\boldsymbol{u}_k$$

$$= -\left(1 - \sigma(\boldsymbol{u}_o^T \boldsymbol{v}_c)\right)\boldsymbol{u}_o + \sum_{k=1}^{K}\left(1 - \sigma(-\boldsymbol{u}_k^T \boldsymbol{v}_c)\right)\boldsymbol{u}_k$$

$$= (\sigma(\boldsymbol{u}_o^T \boldsymbol{v}_c) - 1)\boldsymbol{u}_o + \sum_{k=1}^{K} \sigma(\boldsymbol{u}_k^T \boldsymbol{v}_c)\boldsymbol{u}_k$$

We need to vectorize the 2nd term for improving the performance of computation:

Denote $\boldsymbol{U}_{ng} \in \mathbb{R}^{d\times K}$ as the matrix with column vectors as the *K* word vectors $\boldsymbol{u}_k$ of negative sampled words, *k = 1..., K*. The set of column vectors of $\boldsymbol{U}_{ng}$ is a very small subset of that of $\boldsymbol{U}$

Now, we can rewrite the above-mentioned derivative in a vectorized form:

$$\frac{\partial L_{ng}}{\partial \boldsymbol{v}_c} = (\sigma(\boldsymbol{u}_o^T \boldsymbol{v}_c) - 1)\boldsymbol{u}_o + \boldsymbol{U}_{ng}\, \sigma\left(\boldsymbol{U}_{ng}^T \boldsymbol{v}_c\right)$$

Where $\boldsymbol{U}_{ng}^T \boldsymbol{v}_c$ is a column vector of $\mathbb{R}^K$, and $\sigma\left(\boldsymbol{U}_{ng}^T \boldsymbol{v}_c\right)$ is element-wise sigmoid and it is a column vector of the same dimension as $\boldsymbol{U}_{ng}^T \boldsymbol{v}_c$, numpy's sigmoid supports such element-wise operation. Now the 2nd term of the above-mentioned formula is expressed purely as vector and matrix operations, it is trivial to prove:

$$\boldsymbol{U}_{ng}\, \sigma\left(\boldsymbol{U}_{ng}^T \boldsymbol{v}_c\right) = \sum_{k=1}^{K} \sigma(\boldsymbol{u}_k^T \boldsymbol{v}_c)\boldsymbol{u}_k$$

4. Compute the partial derivative of $L_{ng}(\boldsymbol{v}_c, o, \boldsymbol{U})$ with respect to $\boldsymbol{u}_o$

   Because $o \notin \{w_1, w_{2,\dots}\, w_K\}$, so the derivative of the 2nd term in L_{ng} wrt to $\boldsymbol{u}_o$ *is* **0,** therefore

$$\frac{\partial L_{ng}}{\partial \boldsymbol{u}_o} = -\frac{\sigma(\boldsymbol{u}_o^T \boldsymbol{v}_c)\left(1 - \sigma(\boldsymbol{u}_o^T \boldsymbol{v}_c)\right)}{\sigma(\boldsymbol{u}_o^T \boldsymbol{v}_c)}\boldsymbol{v}_c = (\sigma(\boldsymbol{u}_o^T \boldsymbol{v}_c) - 1)\boldsymbol{v}_c$$

5. Compute the partial derivative of $L_{ng}(\boldsymbol{v}_c, o, \boldsymbol{U})$ with respect to $\boldsymbol{u}_k's$

   Because $o \notin \{w_1, w_{2,\dots}\, w_K\}$, so the derivative of the 1st term in L_{ng} wrt to $\boldsymbol{u}_k$ is **0**

   Case 1: the K negative sampled words w_1, w_2, ..., w_K are distinct

$$\frac{\partial L_{ng}}{\partial \boldsymbol{u}_k} = \frac{\sigma(-\boldsymbol{u}_k^T \boldsymbol{v}_c)\left(1 - \sigma(-\boldsymbol{u}_k^T \boldsymbol{v}_c)\right)}{\sigma(-\boldsymbol{u}_k^T \boldsymbol{v}_c)}\boldsymbol{v}_c = \left(1 - \sigma(-\boldsymbol{u}_k^T \boldsymbol{v}_c)\right)\boldsymbol{v}_c = \sigma(\boldsymbol{u}_k^T \boldsymbol{v}_c)\boldsymbol{v}_c$$

Case 2: the K negative sampled words are not distinct, denote K_0 as the number of distinct words in the K sampled words ($K_0 < K$), w_1, w_2, ..., w_{K_0} are the K_0 distinct sampled words; denote n_k as the number of times each of the K_0 distinct words are sampled, $\sum_{k=1}^{K_0} n_k = K$. Then $L_{ng}(\boldsymbol{v}_c, o, \boldsymbol{U})$ can be rewritten as:

$$L_{ng}(\boldsymbol{v}_c, o, \boldsymbol{U}) = -\log\left(\sigma(\boldsymbol{u}_o^T \boldsymbol{v}_c)\right) - \sum_{k=1}^{K_0} n_k \log\left(\sigma(-\boldsymbol{u}_k^T \boldsymbol{v}_c)\right)$$

Therefore, $\frac{\partial L_{ng}}{\partial \boldsymbol{u}_k}$ is very similar to case 1 (only with an extra n_k):

$$\frac{\partial L_{ng}}{\partial \boldsymbol{u}_k} = n_k \, \sigma(\boldsymbol{u}_k^T \boldsymbol{v}_c) \boldsymbol{v}_c$$

6. Suppose the center word is $c = w_t$ and the context window is $[w_{t-m}, \dots, w_{t-1}, w_t, w_{t+1}, \dots., w_{t+m}\ ]$, where m is the context window size. The total loss for skip-gram version of word2vec for the context window is:

$$J(\boldsymbol{v}_c, w_{t-m}, \dots., w_{t+m}, \boldsymbol{U}) = \sum_{\substack{-m \le j \le m \\ j \ne 0}} L\left(\boldsymbol{v}_c, w_{t+j}, \boldsymbol{U}\right)$$

Where $L\left(\boldsymbol{v}_c, w_{t+j}, \boldsymbol{U}\right)$ could be loss function of the naïve-sofmax $L_{ng}\left(\boldsymbol{v}_c, w_{t+j}, \boldsymbol{U}\right)$ or the negative sampling $L_{ng}\left(\boldsymbol{v}_c, w_{t+j}, \boldsymbol{U}\right)$. Now we have

$$\frac{\partial J(\boldsymbol{v}_c, w_{t-m}, \dots., w_{t+m}, \boldsymbol{U})}{\partial \boldsymbol{U}} = \sum_{\substack{-m \le j \le m \\ j \ne 0}} \frac{\partial L\left(\boldsymbol{v}_c, w_{t+j}, \boldsymbol{U}\right)}{\partial \boldsymbol{U}}$$

$$\frac{\partial J(\boldsymbol{v}_c, w_{t-m}, \dots., w_{t+m}, \boldsymbol{U})}{\partial \boldsymbol{v}_c} = \sum_{\substack{-m \le j \le m \\ j \ne 0}} \frac{\partial L\left(\boldsymbol{v}_c, w_{t+j}, \boldsymbol{U}\right)}{\partial \boldsymbol{v}_c}$$

$$\frac{\partial J(\boldsymbol{v}_c, w_{t-m}, \dots., w_{t+m}, \boldsymbol{U})}{\partial \boldsymbol{v}_w} = \sum_{\substack{-m \le j \le m \\ j \ne 0}} \frac{\partial L\left(\boldsymbol{v}_c, w_{t+j}, \boldsymbol{U}\right)}{\partial \boldsymbol{v}_w} \quad \text{for } w \ne c$$

Sketch of skip-gram algorithm:

With all of the gradients (derivatives) derived in the above-mentioned sections, we can implement a stochastic gradient descent (SGD) algorithm with numpy's matrix and vector operations utilities. At each iteration of the SGD algorithm, we sample a small number of center words, such 32 or 64, and a context window with a certain window size for each center word, calculate the gradients as in the above-mentioned step 6, and then update $\boldsymbol{U}$, $\boldsymbol{V}$ with the gradients and learning rate α, the following is the pseudo-code:

```
Initialize $\boldsymbol{U}$, $\boldsymbol{V}$ to some random values
for i in range (max_num_iterations)
    Sample 32 center words and their context windows of a given window size
    Calculate the loss $J$ and gradients $\frac{\partial J}{\partial \boldsymbol{U}}$, $\frac{\partial J}{\partial \boldsymbol{V}}$
    Update $\boldsymbol{U} = \boldsymbol{U} - \alpha \frac{\partial J}{\partial \boldsymbol{U}}$, $\boldsymbol{V} = \boldsymbol{V} - \alpha \frac{\partial J}{\partial \boldsymbol{V}}$
```