word2vec

Into the search of usable word vectors

Natural Language Processing

Piotr Fulmański

FACULTY OF MATHEMATICS
AND COMPUTER SCIENCE
University of Lodz

Lecture goals

- Sigmoid function, logistic function, softmax function
- Algorithmic method for searching a minimum of a function
- Artificial neural networks - fast introduction
- Interactive examples
- Practical examples

Frequency-based embedding

What we have been doing so far is know as *word embeddings* (or *word vectors* or *word vector representations*). Generally speaking, ***word embeddings* are the texts converted into numbers**.

So far we have seen deterministic methods to determine word vectors:

- one-hot encoding,
- BOW,
- TF-IDF,
- LDA,
- LSA,
- COM.

All of them belongs to so called *frequency-based embedding*.

Prediction-based embedding

There are generally two types of vectors (methods of calculating them) that we encounter under this category:

- Continuous Bag of Words (CBOW),
- Skip-Gram.

These methods are prediction based in the sense that they provided probabilities to the words. This probability allow us to find the most probable word which depending on the **context** (surrounding words).

The undeniable pros of this representation is its usability. This is the first method that allows to do math on words. It may be hard to believe but thanks to these methods we can make computation on words like

```
King - man + woman
```

and as a result we should get

```
Queen
```

Combination of both techniques give us one algorithm known under the name **word2vec**. Both of these are shallow neural networks which map word(s) to the target variable which is also a word(s). Weights of each neural network act as word vector representations.

word2vec

word2vec vc. COM (co-occurence matrix)

We can think of word2vec as a more advanced version of co-occurence matrix. The goal was to create **usable system that creates vectors that are similar to one another for similar words**.

For example, **the method should learn that *backpack* and *crampons* are somewhat similar because they appear near words like *climbing* and *mountain* in sentences. What is important not only *backpack* should be close to *crampons* but also both *backpack* and *crampons* should be closer to *climbing* and *mountain* than to *car* and *driving*.**

Word2vec's approach is to train a models that **predicts** all of the neighboring words for every occurrence of every center word in an entire body of text (a corpus). We refer to each nearby word as the ***context word*** and each word that we are focusing on as the ***center word***. Of course, as it was in case of COM (co-occurence matrix) method, *nearby* actually means words from some *window* of a predefined size (typicaly 5 or 10, which means 5 (10) words behind and 5 (10) words ahead - 10 (20) in total).

word2vec

Continuous Bag of Words vs. Skip-Gram

Both models are opposite to each other. This applies to both their structure and the purpose for which they serve (the task they can solve).

Continuous Bag of Words (CBOW) model try to predict all center word for every context words in the full body of text. It can be thought of as learning word embeddings by training a **model to predict a word given its context**.

Skip-Gram model try to predict all context words for every occurrence of every center word in the full body of text. It can be thought of as learning word embeddings by training a **model to predict context given a word**.

word2vec

Both models are actually surprisingly simple in its most basic form.

Both are shallow neural networks which map word(s) to the target variable which is also a word(s). Weights of each neural network act as *word vector representations*.

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. Then we're not actually going to use that neural network for the task we trained it on. Instead, the weights of the hidden layer are used as some parameters or codes for other process or algorithm which is a real task solver. In our case these weights are the *word vectors* we are looking for.

Example of this approach we have seen in image compression with neural networks. Generally speaking this is used in unsupervised feature learning, where you train an *auto-encoder* to code (compress) an input vector in the hidden layer, and decode (decompress) it back to the original in the output layer.

In some cases, as it is for word2vec, after training, you strip off the output layer and just use such a distorted network consisting only of the hidden layer taken from the initial network.

This approach is very useful as it allows to learn quite good object representation with features without having labeled training data.

Skip-gram model

Skip-gram model

The idea: learning patterns

Our goal is to train the neural network to do the following. Given a specific word in the middle of a sentence (the center word which becomes our ***input word***), the network is going to tell us the probability for every word in our vocabulary of being the ***nearby word***.

We are going to train the neural network so we need some training data.

Consider the below example for very simple corpus consisting of only one document (which in fact is a single sentence). The word highlighted in red is the input word while blue words are nearby words. For simplicity I have used a small window of size 2.

```
corpus = ["Those who know nothing of foreign languages know nothing of their own."]
words = [['those', 'who', 'know', 'nothing', 'of', 'foreign', 'languages', 'know', 'nothing',
'of', 'their', 'own']]
```

```
SOURCE TEXT:                                                             PAIRS

those who know nothing of foreign languages know nothing of their own    (those, who)
                                                                         (those, know)
those who know nothing of foreign languages know nothing of their own    (who, those)
                                                                         (who, know)
                                                                         (who, nothing)
those who know nothing of foreign languages know nothing of their own    (know, those)
                                                                         (know, who)
                                                                         (know, nothing)
                                                                         (know, of)
those who know nothing of foreign languages know nothing of their own    (nothing, who)
                                                                         (nothing, know)
                                                                         (nothing, of)
                                                                         (nothing, foreign)
those who know nothing of foreign languages know nothing of their own    (of, know)
                                                                         (of, nothing)
                                                                         (of, foreign)
                                                                         (of, languages)
```

Skip-gram model

The idea: learning patterns

Every set of pairs generate one training pair where words are encoded with **one-hot encoding**.

Consider the following set of pairs

```
those who know nothing of foreign languages know nothing of their own    [(nothing, who)
                                                                          (nothing, know)
                                                                          (nothing, of)
                                                                          (nothing, foreign)]
```

Our vocabulary is

```
foreign know languages nothing of own their those who
```

We have two options (in this example we use **window of size 2**):

1. Code center word and every context word as a separate one-hot vector (where *t* is the position (index) of a central word in a sentence)

```
nothing => x      = w(t)   = [0,0,0,1,0,0,0,0,0]
who     => y_{-2}= w(t-2) = [0,0,0,0,0,0,0,0,1]
know    => y_{-1}= w(t-1) = [0,1,0,0,0,0,0,0,0]
of      => y_{1} = w(t+1) = [0,0,0,0,1,0,0,0,0]
foreign => y_{2} = w(t+2) = [1,0,0,0,0,0,0,0,0]
```

In this case training pair is of the form

```
(input, target) = (x, (y_{-2},y_{-1},y_{1},y_{2}))
```

2. Combine all context vectors into one (so center word is encoded as one-hot vector while context vectors are encoded as BOW (bag-of words) vector)

```
nothing => x      = w(t)   = [0,0,0,1,0,0,0,0,0]
who     => y_{-2}= w(t-2) = [0,0,0,0,0,0,0,0,1]
know    => y_{-1}= w(t-1) = [0,1,0,0,0,0,0,0,0]
of      => y_{1} = w(t+1) = [0,0,0,0,1,0,0,0,0]
foreign => y_{2} = w(t+2) = [1,0,0,0,0,0,0,0,0]
BOW     => y      =          [1,1,0,0,1,0,0,0,1]
```

In this case training pair is of the form

```
(input, target) = (x, y)
```

Skip-gram model

The idea: learning patterns

Every set of pairs generate one training pair where words are encoded with **one-hot encoding**.

Consider the following set of pairs

```
those who know nothing of foreign languages know nothing of their own     [(nothing, who)
                                                                           (nothing, know)
                                                                           (nothing, of)
                                                                           (nothing, foreign)]
```

Our vocabulary is

```
foreign know languages nothing of own their those who
```

We have two options (in this example we use **window of size 2**):

1. Code center word and every context word as a separate one-hot vector (where *t* is the position (index) of a central word in a sentence)

```
nothing => x      = w(t)   = [0,0,0,1,0,0,0,0,0]
who     => y_{-2}= w(t-2) = [0,0,0,0,0,0,0,0,1]
know    => y_{-1}= w(t-1) = [0,1,0,0,0,0,0,0,0]
of      => y_{1} = w(t+1) = [0,0,0,0,1,0,0,0,0]
foreign => y_{2} = w(t+2) = [1,0,0,0,0,0,0,0,0]
```

This should explains, why the name of this model is skip-gram. We have an n-gram (in our case 5-gram) but we skip one of its element, so we have n-gram with one element left.

In this case training pair is of the form

```
(input, target) = (x, (y_{-2},y_{-1},y_{1},y_{2}))
```

2. Combine all context vectors into one (so center word is encoded as one-hot vector while context vectors are encoded as BOW (bag-of words) vector)

```
nothing => x      = w(t)   = [0,0,0,1,0,0,0,0,0]
who     => y_{-2}= w(t-2) = [0,0,0,0,0,0,0,0,1]
know    => y_{-1}= w(t-1) = [0,1,0,0,0,0,0,0,0]
of      => y_{1} = w(t+1) = [0,0,0,0,1,0,0,0,0]
foreign => y_{2} = w(t+2) = [1,0,0,0,0,0,0,0,0]
BOW     => y      =          [1,1,0,0,1,0,0,0,1]
```

In this case training pair is of the form

```
(input, target) = (x, y)
```

Skip-gram model

The idea: learning patterns

Every set of pairs generate one training pair where words are encoded with **one-hot encoding**.

Consider the following set of pairs

```
those who know nothing of foreign languages know nothing of their own     [(nothing, who)
                                                                           (nothing, know)
                                                                           (nothing, of)
                                                                           (nothing, foreign)]
```

Our vocabulary is

```
foreign know languages nothing of own their those who
```

We have two options (in this example we use **window of size 2**):

1. Code center word and every context word as a separate one-hot vector (where *t* is the position (index) of a central word in a sentence)

```
nothing => x      = w(t)   = [0,0,0,1,0,0,0,0,0]
who     => y_{-2}= w(t-2) = [0,0,0,0,0,0,0,0,1]
know    => y_{-1}= w(t-1) = [0,1,0,0,0,0,0,0,0]
of      => y_{1} = w(t+1) = [0,0,0,0,1,0,0,0,0]
foreign => y_{2} = w(t+2) = [1,0,0,0,0,0,0,0,0]
```

This should explains, why the name of this model is skip-gram. We have an n-gram (in our case 5-gram) but we skip one of its element, so we have n-gram with one element left.

In this case training pair is of the form

```
(input, target) = (x, (y_{-2},y_{-1},y_{1},y_{2}))
```

2. Combine all context vectors into one (so center word is encoded as one-hot vector while context vectors are encoded as BOW (bag-of words) vector)

```
nothing => x      = w(t)   = [0,0,0,1,0,0,0,0,0]
who     => y_{-2}= w(t-2) = [0,0,0,0,0,0,0,0,1]
know    => y_{-1}= w(t-1) = [0,1,0,0,0,0,0,0,0]
of      => y_{1} = w(t+1) = [0,0,0,0,1,0,0,0,0]
foreign => y_{2} = w(t+2) = [1,0,0,0,0,0,0,0,0]
BOW     => y      =          [1,1,0,0,1,0,0,0,1]
```

Notice that one-hot vectors have a lot of components (usually thousands or millions) as our corpus has a lot of words.

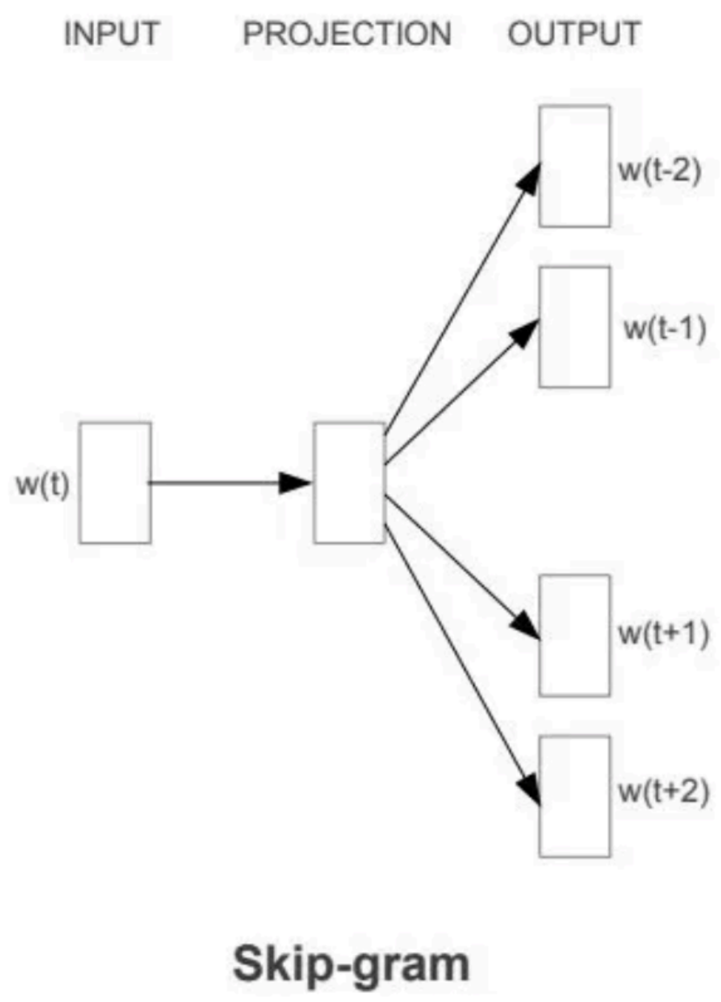
In this case training pair is of the form

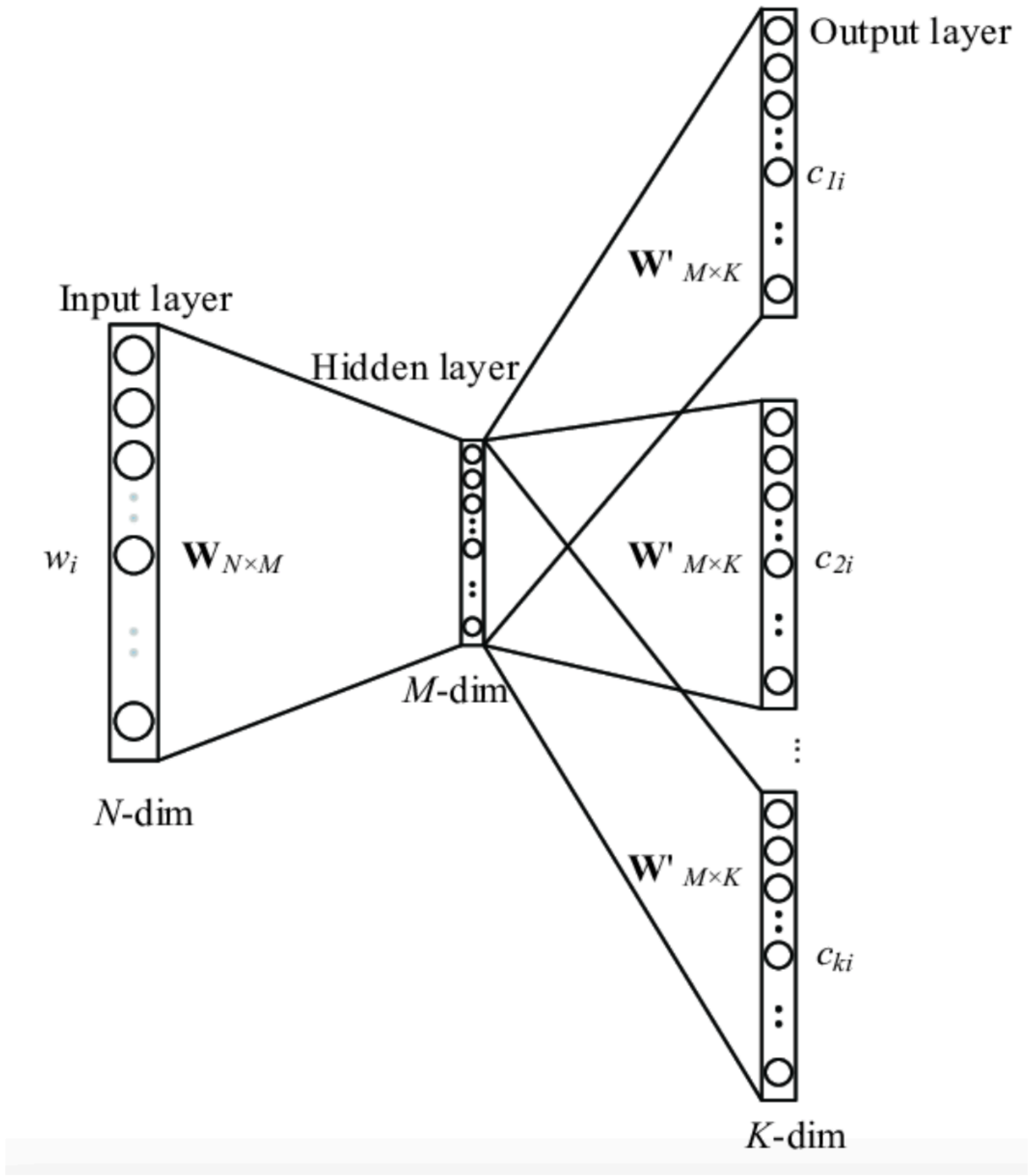
```
(input, target) = (x, y)
```

Skip-gram model

The idea: network architecture

As it was announced, we use a **simple neural network with a single hidden layer**.

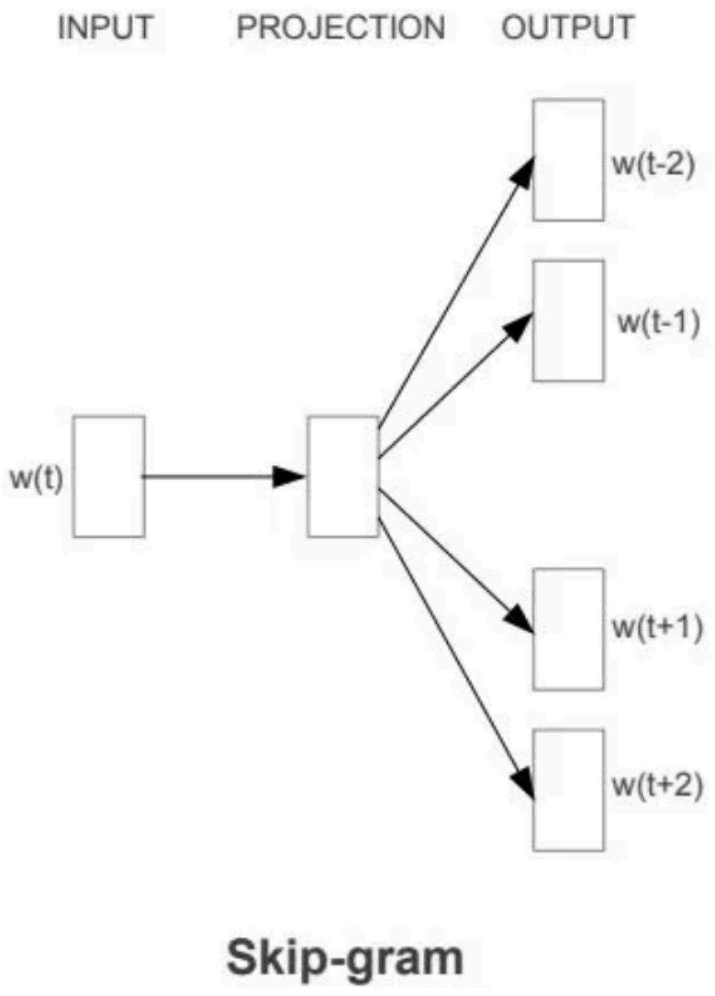


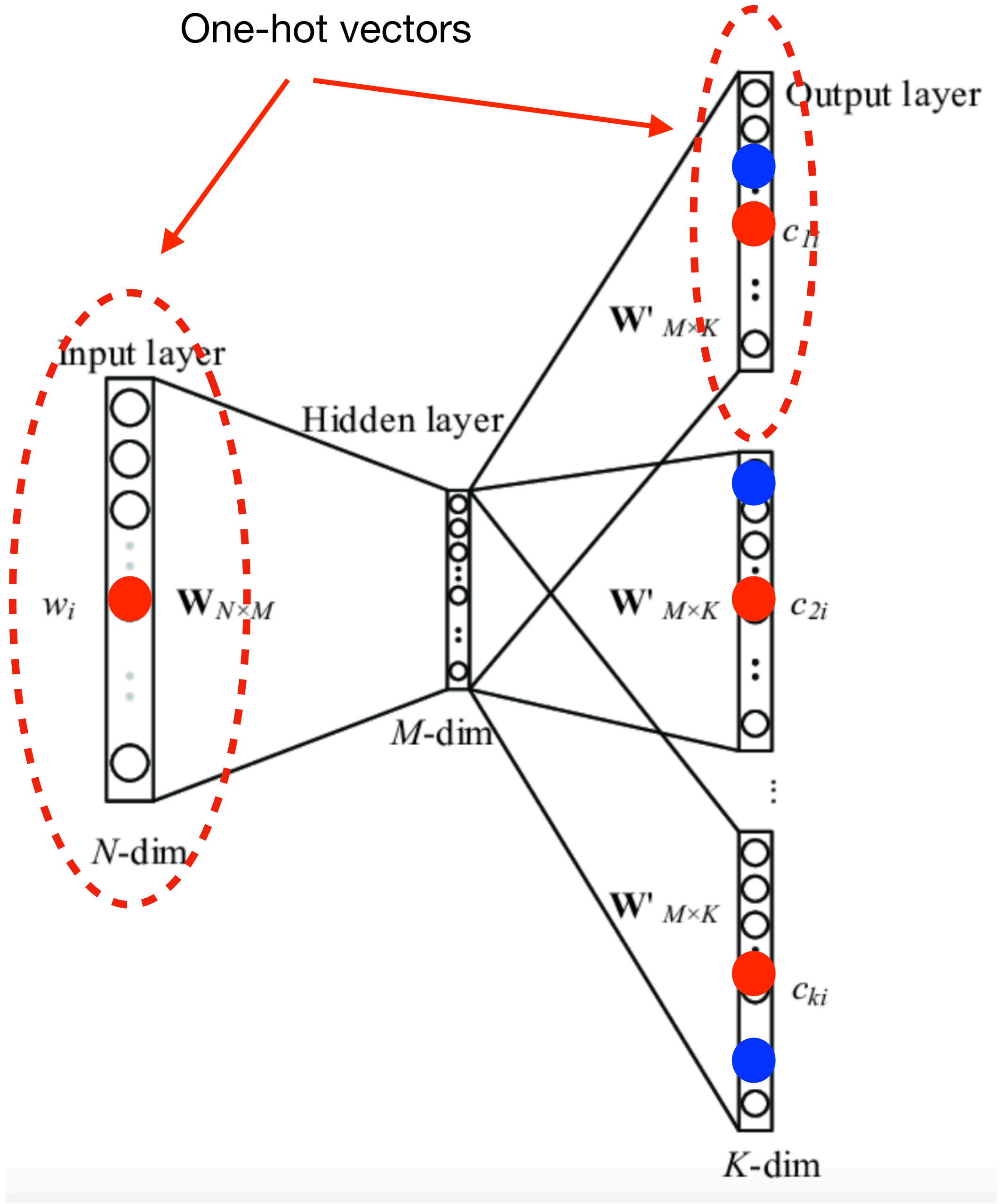


Skip-gram model

The idea: network architecture

As it was announced, we use a **simple neural network with a single hidden layer.**

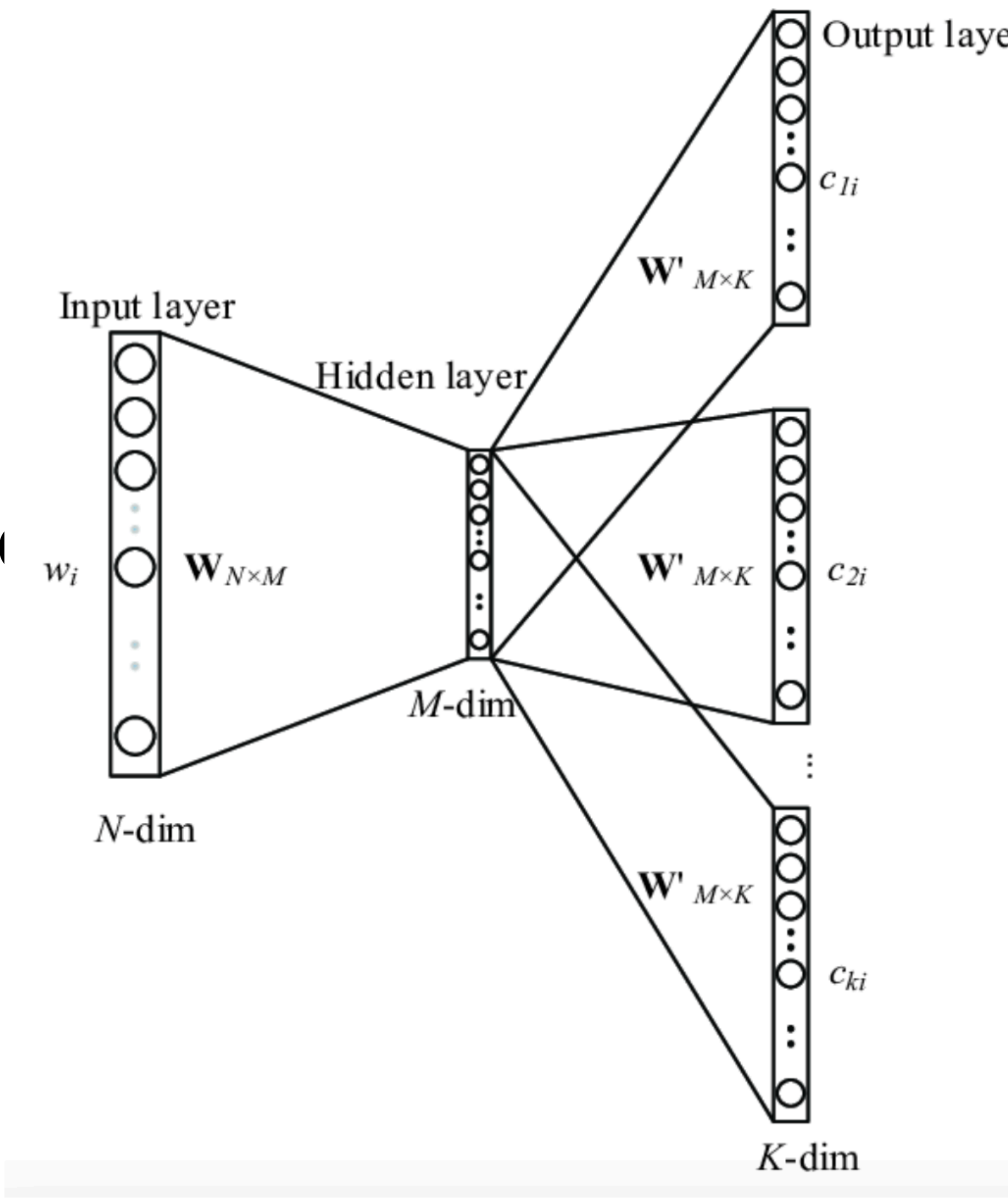




Skip-gram model

The idea: network architecture

In this model we have three layers: **input** layer, **hidden** layer and **output** layer with dense connection - every neuron is connected with every neuron l preceding and/or following layer.



Input layer

This is not true layer as there are no neurons in this layer. We can think of it as a receptors which accept some signals and simply pass them to the subsequent layers.

In this case input is a vector with N components, each is 0 and 1, and representing one central word encoded in one-hot vector form. N is the size of our vocabulary (number of different words in our corpus).

Skip-gram model

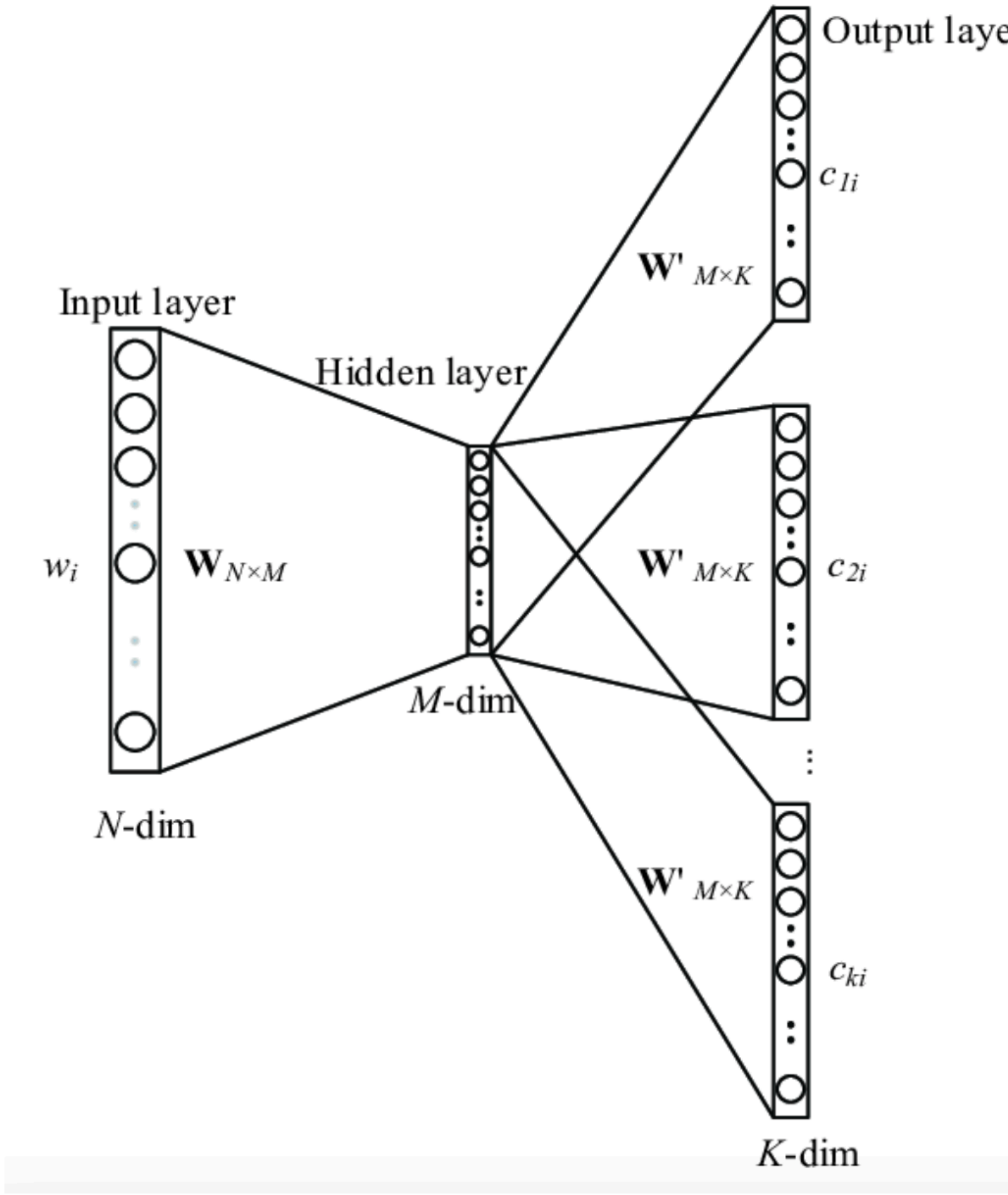
The idea: network architecture

Hidden layer

In this layer simple neurons with **linear activation function** are used.

Dimension of this layer is M, where $M \ll N$.

Output from this layer is our word vector representation (word embedding).



Skip-gram model

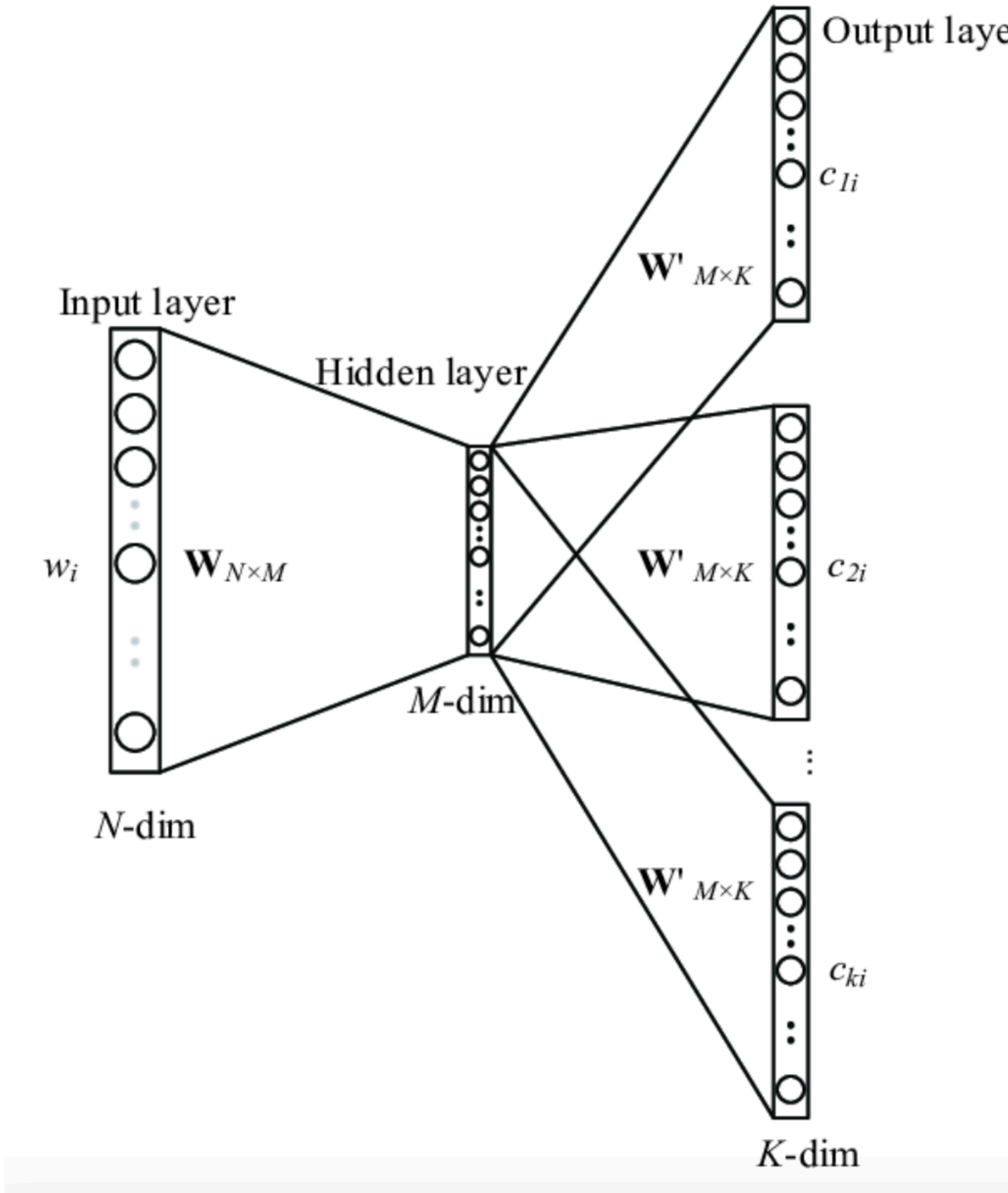
The idea: network architecture

Output layer

In this layer neurons with **sofmax function** are used.

Dimension of this layer may be one of the following

- $N \cdot 2k$, where k is the *window size*. This is a case for output represented as separate one-hot vector.
  See comment on the next slide.



- N - this is a case for output represented as BOW (many one-hot vectors combined into one vector) with exactly $2k$ 1's (and all others are 0's).

Skip-gram model

The idea: network architecture

Output layer - comment to a first case (and also a second)

Define

$$c \in C = \{-k, \ldots, -2, -1, 1, 2, \ldots, k\}$$

Other words c is every integer satisfying

$$-k \leq c \leq k, \; c \neq 0.$$

Then the equations of forward data flow in this model are as follow

$$h = Wx$$

$$u_c = W'h$$

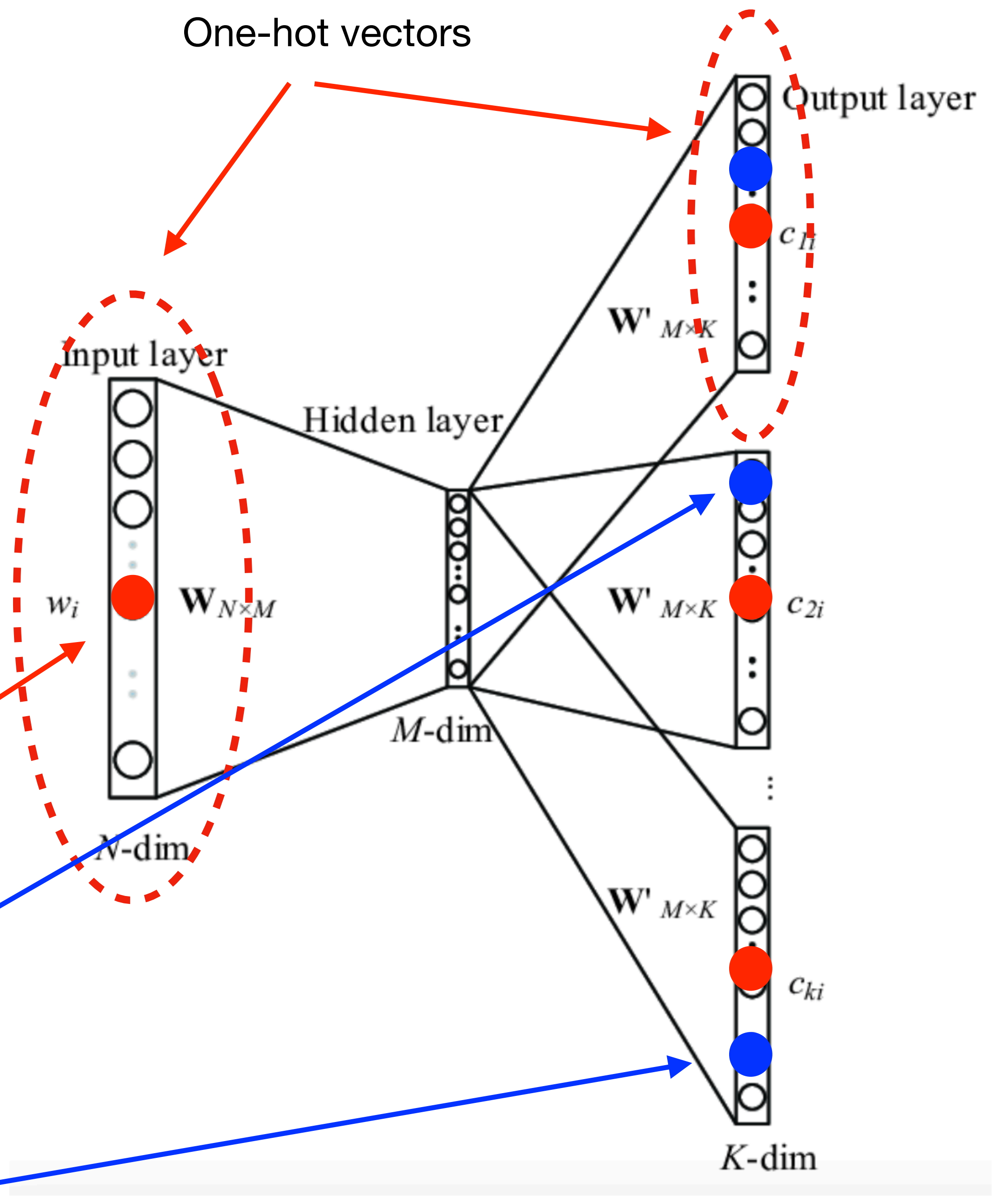
$$y_c = \text{softmax}(u_c)$$

where

$$y_{c,j} = \text{softmax}(u_c)_j = \frac{e^{u_{c,j}}}{\sum_{i=1}^{N} e^{u_{c,i}}}.$$

As we can see the output vectors y_c, are all identical because all are calculated with the same matrix W' and vector h

$$y_c = \text{softmax}(u_c) = \text{softmax}(W'h).$$



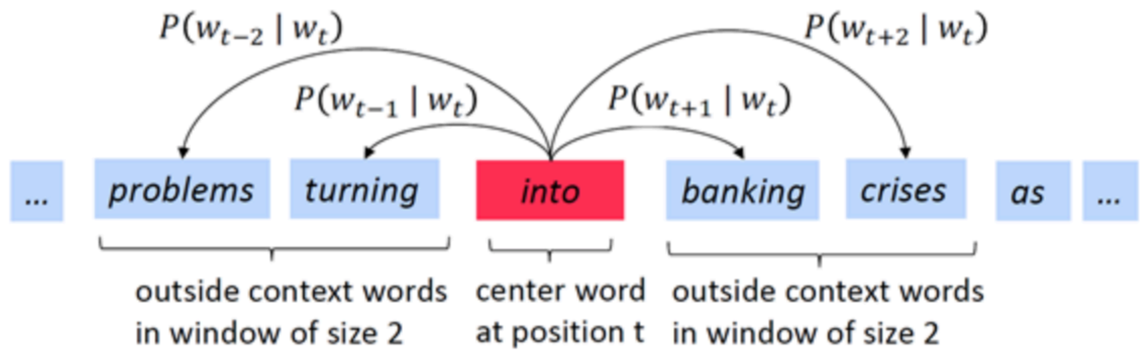
Skip-gram model

The idea: training cycle

To be able to calculate weights updates we have to define an error function and then calculate required derivatives.

Our **goal is to maximize the likelihood of the context words given the center word**, i.e. we will calculate the probability of our model predicting the context words w_{t+c} given the center word w_t and **we will try to maximize that probability**. This likelihood can be represented using this formula:

$$L(\theta) = \prod_{c \in C} P(w_{t+c} \mid w_t; \theta)$$



where

$$\theta = \{W, W'\}$$

is a **set of weights** and

$$P(w_{t+c} \mid w_t; \theta) = \frac{e^{u_{c,t^*}}}{\sum_{i=1}^{N} e^{u_{c,i}}}$$

is a **conditional probabilities of the context word** w_{t+c}**, given the center word** w_t, where t_c^* is an index of word w_{t+c} in your one-hot vector representation, and N is a total number of all different words (the size of our vocabulary).

To put this equation in a form such that it is easy to take derivatives and make it a **minimization problem**, we will just take the log of the equation and multiply it by -1. Note that now multiplication will turn to summation as we have taken the log

$$J(\theta) = -\log L(\theta) = -\sum_{c \in C} \log P(w_{t+c} \mid w_t; \theta)$$

Skip-gram model

The idea: training cycle

Substituting

$$P(w_{t+c}|w_t;\theta) = \frac{e^{u_{c,t_c^*}}}{\sum_{i=1}^{N} e^{u_{c,i}}},$$

to

$$J(\theta) = -\log L(\theta) = -\sum_{c \in C} \log P(w_{t+c}|w_t;\theta)$$

we obtain

$$J(\theta) = -\sum_{c \in C} \log \frac{e^{u_{c,t^*}}}{\sum_{i=1}^{N} e^{u_{c,i}}} = -\sum_{c \in C} (\log e^{u_{c,t_c^*}} - \log \sum_{i=1}^{N} e^{u_{c,i}}) =$$

$$-\sum_{c \in C} u_{c,t_c^*} + \sum_{c \in C} \log \sum_{i=1}^{N} e^{u_{c,i}}$$

Having error (or loss) function defined we can calculate derivatives with respect to weight W'_{ab} in hidden-output layer and with respect to weight W_{ab} in input-hidden layer.

Notation W_{ab} denotes weight between neuron a from layer n and neuron b from layer $n+1$ (we will use this notation)**.**

There is also alternative notation, very popular: where W_{ab} denotes weight between neuron a from layer $n+1$ and neuron b from layer n.

Skip-gram model

The idea: training cycle

Having error (or loss) function defined

$$J(\theta) = -\sum_{c \in C} u_{c,t_c^*} + \sum_{c \in C} \log \sum_{i=1}^{N} e^{u_{c,i}}$$

we can calculate derivatives with respect to weight W'_{ab} in hidden-output layer and with respect to weight W_{ab} in input-hidden layer.

It is clear that this function depends on the weights W and W' through the $2k$ (because we have $2k$ output vectors u) variables u_c each with N components of the form $(u_{c,1}, \ldots, u_{c,N})$.

The derivatives then simply follow from the chain rule for multivariate functions,

$$\frac{\partial J}{\partial W'_{ab}} = \sum_{k=1}^{N} \sum_{c \in C} \frac{\partial J}{\partial u_{c,k}} \frac{\partial u_{c,k}}{\partial W'_{ab}}, a = 1,\ldots,M, b = 1,\ldots,N$$

$$\frac{\partial J}{\partial W_{ab}} = \sum_{k=1}^{N} \sum_{c \in C} \frac{\partial J}{\partial u_{c,k}} \frac{\partial u_{c,k}}{\partial W_{ab}}, a = 1,\ldots,N, b = 1,\ldots,M.$$

Skip-gram model

The idea: training cycle

For error (or loss) function of the form

$$J(\theta) = -\sum_{c\in C} u_{c,t_c^*} + \sum_{c\in C} \log \sum_{i=1}^{N} e^{u_{c,i}}$$

we can calculate

$$\frac{\partial J}{\partial u_{c,k}} = -\delta_{kt_c^*} + \left(\log \sum_{i=1}^{N} e^{u_{-k,i}}\right)' + \ldots + \left(\log \sum_{i=1}^{N} e^{u_{c,i}}\right)' \ldots + \left(\log \sum_{i=1}^{N} e^{u_{k,i}}\right)' =$$

$$-\delta_{kt_c^*} + \frac{1}{\sum_{i=1}^{N} e^{u_{-k,i}}} \sum_{i=1}^{N} \left(e^{u_{-k,i}}\right)' + \ldots + \frac{1}{\sum_{i=1}^{N} e^{u_{c,i}}} \sum_{i=1}^{N} \left(e^{u_{c,i}}\right)' \ldots + \frac{1}{\sum_{i=1}^{N} e^{u_{k,i}}} \sum_{i=1}^{N} \left(e^{u_{k,i}}\right)' =$$

$$-\delta_{kt_c^*} + \frac{1}{\sum_{i=1}^{N} e^{u_{c,i}}} e^{u_{c,k}} = -\delta_{jt_c^*} + \frac{e^{u_{c,k}}}{\sum_{i=1}^{N} e^{u_{c,i}}} =$$

$$-\delta_{kt_c^*} + y_{c,k}$$

Skip-gram model

The idea: training cycle

$u_{c,k}$ is the k-th component from c context word (one-hot vector).

$$u_{c,k} = \sum_{m=1}^{M} \left(W'_{mk} \cdot h_m\right) = \sum_{m=1}^{M} \left(W'_{mk} \cdot \sum_{n=1}^{N} W_{nm} x_n\right)$$

$$\frac{\partial u_{c,k}}{\partial W'_{ab}} = \left(\sum_{m=1}^{M} \left(W'_{mk} \cdot \sum_{n=1}^{N} W_{nm} x_n\right)\right)' = \delta_{kb} \cdot \sum_{n=1}^{N} W_{na} x_n$$

$$\frac{\partial u_{c,k}}{\partial W_{ab}} = \left(\sum_{m=1}^{M} \left(W'_{mk} \cdot \sum_{n=1}^{N} W_{nm} x_n\right)\right)' = W'_{bk} \cdot x_a$$

Skip-gram model

The idea: training cycle

Finally the derivative of J with respect to W'_{ab} takes the form

$$\frac{\partial J}{\partial W'_{ab}} = \sum_{k=1}^{N}\sum_{c\in C}\frac{\partial J}{\partial u_{c,k}}\frac{\partial u_{c,k}}{\partial W'_{ab}} =$$

$$\sum_{k=1}^{N}\sum_{c\in C}\left(-\delta_{kt_c^*}+y_{c,k}\right)\left(\delta_{kb}\cdot\sum_{n=1}^{N}W_{na}x_n\right) =$$

$$\sum_{k=1}^{N}\sum_{c\in C}\delta_{kb}\left(-\delta_{kt_c^*}+y_{c,k}\right)\left(\sum_{n=1}^{N}W_{na}x_n\right) =$$

$$\sum_{c\in C}\left(-\delta_{bt_c^*}+y_{c,b}\right)\left(\sum_{n=1}^{N}W_{na}x_n\right)$$

Skip-gram model

The idea: training cycle

Finally the derivative of J with respect to W_{ab} takes the form

$$\frac{\partial J}{\partial W_{ab}} = \sum_{k=1}^{N} \sum_{c \in C} \frac{\partial J}{\partial u_{c,k}} \frac{\partial u_{c,k}}{\partial W_{ab}} =$$

$$\sum_{k=1}^{N} \sum_{c \in C} \left(-\delta_{kt_c^*} + y_{c,k} \right) W'_{bk} \cdot x_a$$

Skip-gram model

The idea: training cycle

Finally the derivative of J with respect to W'_{ab} and W_{ab} takes the form

$$\frac{\partial J}{\partial W'_{ab}} = \sum_{c \in C} \left(-\delta_{bt_c^*} + y_{c,b} \right) \left(\sum_{n=1}^{N} W_{na} x_n \right)$$

$$\frac{\partial J}{\partial W_{ab}} = \sum_{k=1}^{N} \sum_{c \in C} \left(-\delta_{kt_c^*} + y_{c,k} \right) W'_{bk} \cdot x_a$$

Vectorize formulas for derivatives

Skip-gram model

Vectorize formulas for derivatives

Outer product

Given two vectors

$$\mathbf{u} = \left(u_1, u_2, \ldots, u_m\right)$$

$$\mathbf{v} = \left(v_1, v_2, \ldots, v_n\right)$$

their outer product, denoted $\mathbf{u} \otimes \mathbf{v}$ is defined as the $m \times n$ matrix A obtained by multiplying each element of $\mathbf{u}$ by each element of $\mathbf{v}$

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

Skip-gram model

Vectorize formulas for derivatives

$$\mathbf{u} = \left(u_1, u_2, \ldots, u_m\right)$$

$$\mathbf{v} = \left(v_1, v_2, \ldots, v_n\right)$$

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

$$\frac{\partial J}{\partial W'_{ab}} = \underbrace{\sum_{c \in C} \left(-\delta_{bt^*} + y_{c,b}\right)}_{A_b} \underbrace{\left(\sum_{n=1}^{N} W_{na} x_n\right)}_{B_a}$$

$$\frac{\partial J}{\partial W'} = \begin{bmatrix} \frac{\partial J}{\partial W'_{11}} & \frac{\partial J}{\partial W'_{12}} & \ldots & \frac{\partial J}{\partial W'_{1N}} \\ \frac{\partial J}{\partial W'_{21}} & \frac{\partial J}{\partial W'_{22}} & \ldots & \frac{\partial J}{\partial W'_{2N}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial J}{\partial W'_{M1}} & \frac{\partial J}{\partial W'_{M2}} & \ldots & \frac{\partial J}{\partial W'_{MN}} \end{bmatrix}$$

$$\frac{\partial J}{\partial W'} = \begin{bmatrix} B_1 A_1 & B_1 A_2 & \ldots & B_1 A_N \\ B_2 A_1 & B_2 A_2 & \ldots & B_2 A_N \\ \vdots & \vdots & \ddots & \vdots \\ B_M A_1 & B_M A_2 & \ldots & B_M A_N \end{bmatrix}$$

$$\mathbf{B} = \left(B_1, B_2, \ldots, B_M\right)$$

$$\mathbf{A} = \left(A_1, A_2, \ldots, A_N\right)$$

$$\frac{\partial J}{\partial W'} = \mathbf{B} \otimes \mathbf{A}$$

Skip-gram model

Vectorize formulas for derivatives

$$\frac{\partial J}{\partial W'_{ab}} = \underbrace{\sum_{c \in C} \left(-\delta_{bt^*} + y_{c,b}\right)}_{A_b} \underbrace{\left(\sum_{n=1}^{N} W_{na} x_n\right)}_{B_a}$$

$$B_a = \left(\sum_{n=1}^{N} W_{na} x_n\right)$$

$$\mathbf{B} = \left(B_1, B_2, \ldots, B_M\right)$$

$$\mathbf{A} = \left(A_1, A_2, \ldots, A_N\right)$$

$$W = \begin{bmatrix} W_{11} & W_{12} & \ldots & W_{1a} & \ldots & W_{1M} \\ W_{21} & W_{22} & \ldots & W_{2a} & \ldots & W_{2M} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ W_{n1} & W_{n2} & \ldots & W_{na} & \ldots & W_{nM} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ W_{N1} & W_{N2} & \ldots & W_{Na} & \ldots & W_{NM} \end{bmatrix}$$

$$\frac{\partial J}{\partial W'} = \begin{bmatrix} B_1 A_1 & B_1 A_2 & \ldots & B_1 A_N \\ B_2 A_1 & B_2 A_2 & \ldots & B_2 A_N \\ \vdots & \vdots & \ddots & \vdots \\ B_M A_1 & B_M A_2 & \ldots & B_M A_N \end{bmatrix}$$

$$\frac{\partial J}{\partial W'} = \mathbf{B} \otimes \mathbf{A}$$

Skip-gram model

Vectorize formulas for derivatives

The most natural way of thinking about vector is to treat it as a 1-D array or a list.

In NumPy we have

```
import numpy as np

# Create a 1-D (horizontal) list
listH = [1, 2, 3]

# Create a 1-D (vertical) list
listV = [[10],
        [20],
        [30]]

# Create a vector based on listH
# This would be row vector
vectorH = np.array(listH)

# Create a vector based on listV
# This would be column vector
vectorV = np.array(listV)
```

```
# Verify it
print("Row (horizontal) vector")
print(vectorH)
print("----------------")
print("Column (vertical)
vector")
print(vectorV)
```

which prints

```
Row (horizontal) vector
[1 2 3]
----------------
Column (vertical) vector
[[10]
 [20]
 [30]]
```

Thats why we may say, that in NumPy vectors are row (horizontal) oriented. **This is really very important when you are close to implementing some mathematical formulas, as we do now.**

Skip-gram model

Vectorize formulas for derivatives

NumPy seems to be very tolerant in case of multiplying matrix and vector.

As you know, to multiply two matrices or matrix by vector (which can be interpreted as 1-D row or column vector) their dimensions should match

$$A_{n,m} \cdot B_{m,k} = C_{n,k}$$

Formally in the following example vector `vectorH` is of the shape 1×2, vector `vectorV` is 2×1 and matrix `m` is 3×2

```
import numpy as np

listH = [1, 2]
listV = [[10],
         [20]]

vectorH = np.array(listH)
vectorV = np.array(listV)

m = np.array([[1, 2],
              [3, 4],
              [5, 6]])

print(vectorH.shape)
print(vectorV.shape)
print(m.shape)

print("--- correct -------------")
x = np.matmul(m, vectorH)
print(x)

print("--- correct -------------")
x = np.matmul(m, vectorV)
print(x)

print("--- incorrect -----------")
x = np.matmul(vectorH, m)
print(x)
```

which prints

```
(2,)
(2, 1)
(3, 2)
--- correct -------------
[ 5 11 17]
--- correct -------------
[[ 50]
 [110]
 [170]]
--- incorrect -----------
Traceback (most recent call last):
  File "<string>", line 21, in <module>
ValueError: matmul: Input operand 1 has a
mismatch in its core dimension 0, with gufunc
signature (n?,k),(k,m?)->(n?,m?) (size 3 is
different from 2)
```

The following multiplication, from mathematical point of view **is not feasible** (because we have [3,2] x [1,2]=???)

```
|1 2|
|3 4|*[1,2]=[1*1+2*2,3*1+4*2,5*1+6*2]=[5,11,17]
|5 6|
```

but this is what NumPy does. The following is correct (because we have [3,2] x [2,1] = [3,1])

```
|1 2|   |10|   |1*10 + 2*20|   | 50|
|3 4| * |20| = |3*10 + 4*20| = |110|
|5 6|          |5*10 + 6*20|   |170|
```

Skip-gram model

Vectorize formulas for derivatives

$$B_a = \left(\sum_{n=1}^{N} W_{na} x_n \right)$$

$$W = \begin{bmatrix} W_{11} & W_{12} & \cdots & W_{1a} & \cdots & W_{1M} \\ W_{21} & W_{22} & \cdots & W_{2a} & \cdots & W_{2M} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ W_{n1} & W_{n2} & \cdots & W_{na} & \cdots & W_{nM} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ W_{N1} & W_{N2} & \cdots & W_{Na} & \cdots & W_{NM} \end{bmatrix}_{N \times M}$$

$$W^T = \begin{bmatrix} W_{11} & W_{21} & \cdots & W_{n1} & \cdots & W_{N1} \\ W_{12} & W_{22} & \cdots & W_{n2} & \cdots & W_{N2} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ W_{1a} & W_{2a} & \cdots & W_{na} & \cdots & W_{Na} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ W_{1M} & W_{2M} & \cdots & W_{nM} & \cdots & W_{NM} \end{bmatrix}_{M \times N}$$

Finally we have

$$\mathbf{B} = \mathbf{W}^T \mathbf{x}$$

Skip-gram model

Vectorize formulas for derivatives

Component

$$A_b = \sum_{c \in C} \left(-\delta_{bt_c^*} + y_{c,b} \right)$$

is an error.

If c-th context word **should** be coded at position b then, if output is **correct** which means that $y_{c,b} = 1$ (c -th context word, encoded as on-hot vector, at position b has correctly 1), we have

- $t_c^* = b$
- $-\delta_{bt_c^*} = -\delta_{bb} = 1$
- $-\delta_{bb} + y_{c,b} = -1 + 1 = 0$

If c-th context word **should** be coded at position b then, if output is **incorrect** which means that $y_{c,b} = 0$ (c-th context word, encoded as on-hot vector, at position b has incorrectly 0), we have

- $t_c^* = b$
- $-\delta_{bt_c^*} = -\delta_{bb} = 1$
- $-\delta_{bb} + y_{c,b} = -1 + 0 = -1$

If c-th context word **shouldn't** be coded at position b then, if output is **correct** which means that $y_{c,b} = 0$ (c -th context word, encoded as on-hot vector, at position b has correctly 0), we have

- $t_c^* \neq b$
- $-\delta_{bt_c^*} = 0$
- $-\delta_{bt_c^*} + y_{c,b} = 0 + 0 = 0$

If c-th context word **shouldn't** be coded at position b then, if output is **incorrect** which means that $y_{c,b} = 1$ (c-th context word, encoded as on-hot vector, at position b has incorrectly 1), we have

- $t_c^* \neq b$
- $-\delta_{bt_c^*} = 0$
- $-\delta_{bt_c^*} + y_{c,b} = 0 + 1 = 1$

Skip-gram model

Vectorize formulas for derivatives

Component

$$A_b = \sum_{c \in C} \left(-\delta_{bt_c^*} + y_{c,b} \right)$$

is an error. We introduce a new symbol e_{cb} as

$$e_{cb} = -\delta_{bt_c^*} + y_{c,b}$$

$$A_b = \sum_{c \in C} e_{c,b}$$

$$\mathbf{A} = (A_1, A_2, \ldots, A_N) = (\sum_{c \in C} e_{c,1}, \sum_{c \in C} e_{c,2}, \ldots, \sum_{c \in C} e_{c,N})$$

So every component of $\mathbf{A}$ is a sum of C values. This is equal to sum of corresponding components of C vectors

$$\mathbf{e}_c = (e_{c1}, e_{c2}, \ldots, e_{cN})$$

$$\mathbf{A} = \sum_{c \in C} \mathbf{e}_c$$

Skip-gram model

Vectorize formulas for derivatives

So far we have the following results

$$\frac{\partial J}{\partial W'} = \mathbf{B} \otimes \mathbf{A}$$

$$\mathbf{B} = \mathbf{W}^T \mathbf{x}$$

$$\mathbf{A} = \sum_{c \in C} \mathbf{e}_c$$

Putting this together we obtain

$$\frac{\partial J}{\partial W'} = \left(\mathbf{W}^T \mathbf{x}\right) \otimes \sum_{c \in C} \mathbf{e}_c$$

Skip-gram model

Vectorize formulas for derivatives

$$\frac{\partial J}{\partial W_{ab}} = \sum_{k=1}^{N} \sum_{c \in C} \left(-\delta_{kt_c^*} + y_{c,k} \right) W'_{bk} \cdot x_a$$

$$\frac{\partial J}{\partial W_{ab}} = \underbrace{\left(\sum_{k=1}^{N} \sum_{c \in C} \left(-\delta_{kt_c^*} + y_{c,k} \right) W'_{bk} \right)}_{A_b} \cdot \underbrace{x_a}_{B_a}$$

$$\frac{\partial J}{\partial W} = \begin{bmatrix} \frac{\partial J}{\partial W_{11}} & \frac{\partial J}{\partial W_{12}} & \cdots & \frac{\partial J}{\partial W_{1M}} \\ \frac{\partial J}{\partial W_{21}} & \frac{\partial J}{\partial W_{22}} & \cdots & \frac{\partial J}{\partial W_{2M}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial J}{\partial W_{N1}} & \frac{\partial J}{\partial W_{N2}} & \cdots & \frac{\partial J}{\partial W_{NM}} \end{bmatrix}$$

$$\frac{\partial J}{\partial W} = \begin{bmatrix} B_1A_1 & B_1A_2 & \dots & B_1A_M \\ B_2A_1 & B_2A_2 & \dots & B_2A_M \\ \vdots & \vdots & \ddots & \vdots \\ B_NA_1 & B_NA_2 & \dots & B_NA_M \end{bmatrix}$$

$$\mathbf{B} = \left(B_1, B_2, \dots, B_M \right)$$

$$\mathbf{A} = \left(A_1, A_2, \dots, A_N \right)$$

$$\frac{\partial J}{\partial W} = \mathbf{B} \otimes \mathbf{A}$$

Skip-gram model

Vectorize formulas for derivatives

$$\frac{\partial J}{\partial W_{ab}} = \underbrace{\left(\sum_{k=1}^{N} \sum_{c \in C} \left(-\delta_{kt_c^*} + y_{c,k} \right) W'_{bk} \right)}_{A_b} \cdot \underbrace{x_a}_{B_a}$$

$$\mathbf{B} = \left(B_1, B_2, \ldots, B_M\right)$$

$$B_a = x_a$$

$$\mathbf{B} = \left(x_1, x_2, \ldots, x_M\right)$$

$$\mathbf{B} = \mathbf{x}$$

$$\frac{\partial J}{\partial W} = \begin{bmatrix} B_1A_1 & B_1A_2 & \ldots & B_1A_M \\ B_2A_1 & B_2A_2 & \ldots & B_2A_M \\ \vdots & \vdots & \ddots & \vdots \\ B_NA_1 & B_NA_2 & \ldots & B_NA_M \end{bmatrix}$$

$$\mathbf{B} = \left(B_1, B_2, \ldots, B_M\right)$$

$$\mathbf{A} = \left(A_1, A_2, \ldots, A_N\right)$$

$$\frac{\partial J}{\partial W} = \mathbf{B} \otimes \mathbf{A}$$

Skip-gram model

Vectorize formulas for derivatives

$$e_{ck} = -\delta_{kt_c^*} + y_{c,k}$$

$$\frac{\partial J}{\partial W_{ab}} = \underbrace{\left(\sum_{k=1}^{N} \sum_{c \in C} \underbrace{\left(-\delta_{kt_c^*} + y_{c,k} \right)}_{e_{c,k}} W'_{bk} \right)}_{A_b} \cdot \underbrace{x_a}_{B_a}$$

$$\frac{\partial J}{\partial W} = \begin{bmatrix} B_1 A_1 & B_1 A_2 & \dots & B_1 A_M \\ B_2 A_1 & B_2 A_2 & \dots & B_2 A_M \\ \vdots & \vdots & \ddots & \vdots \\ B_N A_1 & B_N A_2 & \dots & B_N A_M \end{bmatrix}$$

$$\mathbf{B} = \left(B_1, B_2, \dots, B_M \right)$$

$$\mathbf{A} = \left(A_1, A_2, \dots, A_N \right)$$

$$\frac{\partial J}{\partial W} = \mathbf{B} \otimes \mathbf{A}$$

$$\mathbf{A} = \left(A_1, A_2, \dots, A_N \right)$$

$$A_1 = \sum_{k=1}^{N} \sum_{c \in C} e_{c,k} W'_{1k}$$

$$A_1 = \sum_{c \in C} e_{c,1} W'_{11} + \sum_{c \in C} e_{c,2} W'_{12} + \dots + \sum_{c \in C} e_{c,N} W'_{1N}$$

Skip-gram model

Vectorize formulas for derivatives

$$\mathbf{A} = \left(A_1, A_2, \ldots, A_N\right)$$

$$A_1 = \sum_{k=1}^{N} \sum_{c \in C} e_{c,k} W'_{1k}$$

$$A_1 = \sum_{c \in C} e_{c,1} W'_{11} + \sum_{c \in C} e_{c,2} W'_{12} + \ldots + \sum_{c \in C} e_{c,N} W'_{1N}$$

$$A_b = \sum_{c \in C} e_{c,1} W'_{b1} + \sum_{c \in C} e_{c,2} W'_{b2} + \ldots + \sum_{c \in C} e_{c,N} W'_{bN}$$

$$\begin{array}{cc} \mathbf{C}_1 & \\ & \end{array} \begin{bmatrix} W'_{11} & W'_{12} & \cdots & W'_{1N} \\ W'_{21} & W'_{22} & \cdots & W'_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ W'_{b1} & W'_{b2} & \cdots & W'_{bN} \\ \vdots & \vdots & \ddots & \vdots \\ W'_{M1} & W'_{M2} & \cdots & W'_{MN} \end{bmatrix} \underbrace{\begin{bmatrix} \sum_{c \in C} e_{c,1} \\ \sum_{c \in C} e_{c,2} \\ \vdots \\ \sum_{c \in C} e_{c,N} \end{bmatrix}}_{\mathbf{D}}$$

$$\mathbf{A}_1 = \mathbf{C}_1 \mathbf{D}$$

$$\mathbf{A}_b = \mathbf{C}_b \mathbf{D}$$

$$\mathbf{A} = \mathbf{W}' \mathbf{D}$$

$$\mathbf{e}_c = (e_{c1}, e_{c2}, \ldots, e_{cN})$$

So every component of $\mathbf{D}$ (there are N components) is a sum of C values. This is equal to sum of corresponding components of C vectors $\mathbf{e}_c$ of the form

$$\mathbf{e}_c = (e_{c1}, e_{c2}, \ldots, e_{cN})$$

$$\mathbf{D} = \sum_{c \in C} \mathbf{e}_c$$

$$\mathbf{A} = \mathbf{W}' \sum_{c \in C} \mathbf{e}_c$$

Skip-gram model

Vectorize formulas for derivatives

So far we have the following results

$$\frac{\partial J}{\partial W} = \mathbf{B} \otimes \mathbf{A}$$

$$\mathbf{B} = \mathbf{x}$$

$$\mathbf{A} = \mathbf{W}' \sum_{c \in C} \mathbf{e}_c$$

Putting this together we obtain

$$\frac{\partial J}{\partial W} = \mathbf{x} \otimes \mathbf{W}' \sum_{c \in C} \mathbf{e}_c$$

Skip-gram model

Vectorize formulas for derivatives

$$\frac{\partial J}{\partial W'_{ab}} = \sum_{c \in C} \left(-\delta_{bt_c^*} + y_{c,b} \right) \left(\sum_{n=1}^{N} W_{na} x_n \right)$$

$$\frac{\partial J}{\partial W'} = \left(\mathbf{W}^T \mathbf{x} \right) \otimes \sum_{c \in C} \mathbf{e}_c$$

$$\frac{\partial J}{\partial W_{ab}} = \sum_{k=1}^{N} \sum_{c \in C} \left(-\delta_{kt_c^*} + y_{c,k} \right) W'_{bk} \cdot x_a$$

$$\frac{\partial J}{\partial W} = \mathbf{x} \otimes \mathbf{W}' \sum_{c \in C} \mathbf{e}_c$$

Implementation

Skip-gram model

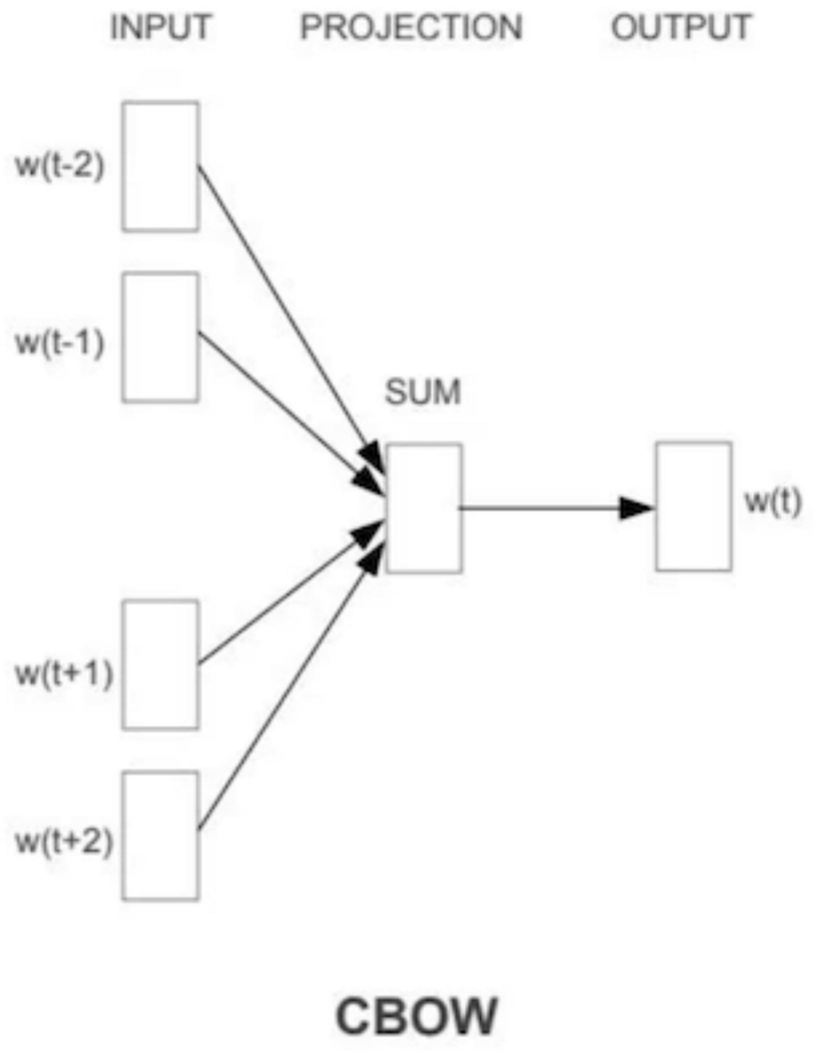
Practical implementation, stage 1:

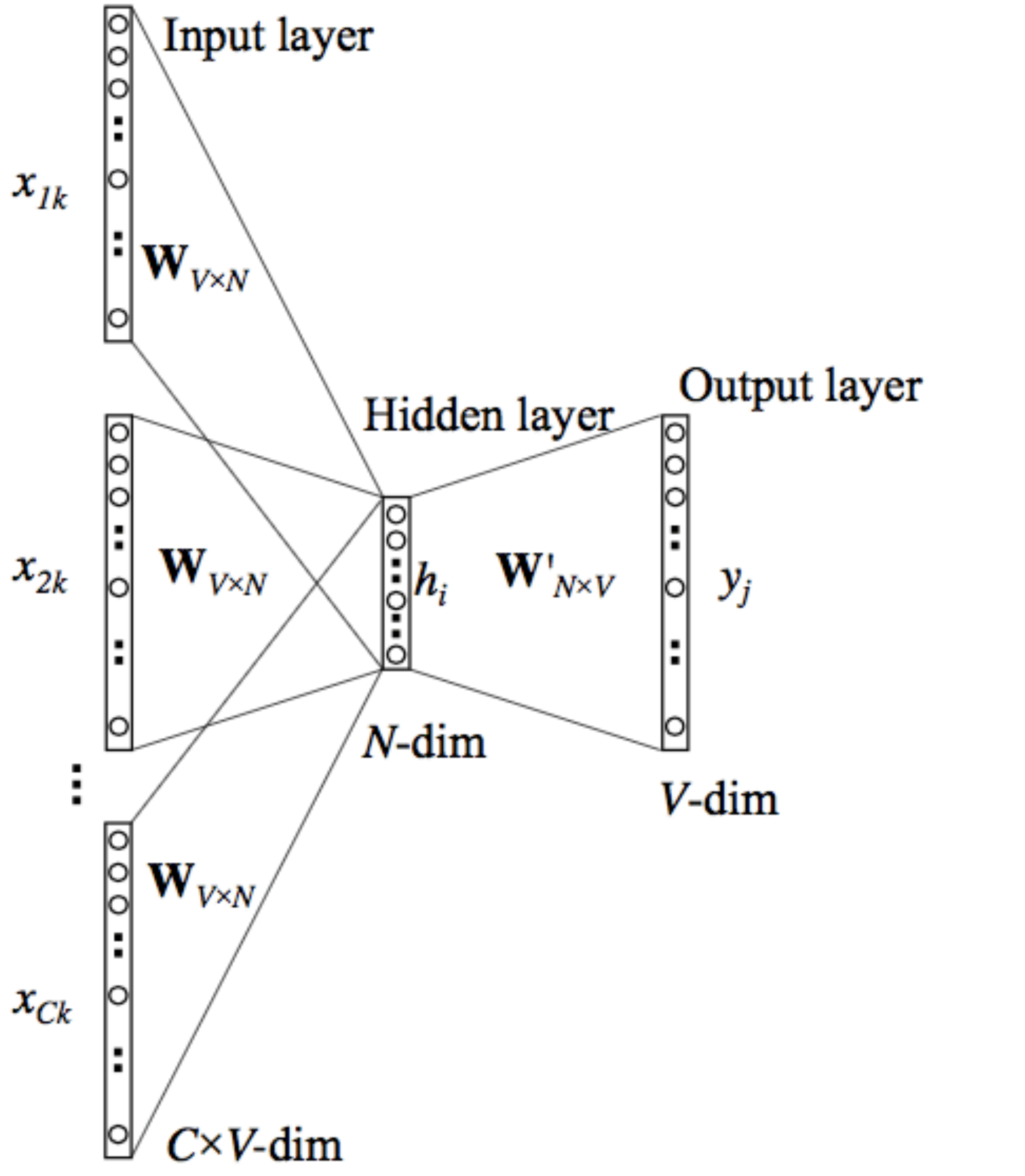
Some notes

CBOW model

The idea: training cycle

The CBOW model is essentially the inverse of the skip-gram model.





Speed-up training

Training of word2vec is a very computationally expensive process. With millions of word the training may take a lot of time. To speed up this process we can apply

- subsampling frequent words,
- negative sampling,
- hierarchical softmax.

Pre-trained

- Example

```
from gensim.models import Word2Vec

# Loading the model.
model = Word2Vec.load_word2vec_format('GoogleNews-vectors-negative300.bin',
binary=True, norm_only=True)

# The model is loaded - now it can be used.
dog = model['dog']

# Performing king queen magic.
print(model.most_similar(positive=['woman', 'king'], negative=['man']))

# Picking odd one out.
print(model.doesnt_match("breakfast cereal dinner lunch".split()))

# Printing similarity index.
print(model.similarity('woman', 'man'))

# Define our own corpus.
sentence=[[‘Neeraj’,’Boy’],[‘Sarwan’,’is’],[‘good’,’boy’]]

# Training word2vec on our corpus.
model = gensim.models.Word2Vec(sentence, min_count=1,size=300,workers=4)

#Using the model. The new trained model can be used similar to the pre-trained
ones.
```

- See pre-trained *word2vec* section in Bibliography.

Bibliography

- Math related to word2vec
    - The backpropagation algorithm for Word2Vec
      `http://www.claudiobellei.com/2018/01/06/backprop-word2vec/`
    - Ankur Tomar, *A math-first explanation of Word2Vec*
      `https://medium.com/analytics-vidhya/maths-behind-word2vec-explained-38d74f32726b`
- Pre-trained word2vec:
    - Usman Malik, *Implementing Word2Vec with Gensim Library in Python*,
      `https://stackabuse.com/implementing-word2vec-with-gensim-library-in-python/`
    - Gensim Word2Vec Tutorial,
      `https://www.kaggle.com/pierremegret/gensim-word2vec-tutorial`
    - Python | Word Embedding using Word2Vec,
      `https://www.geeksforgeeks.org/python-word-embedding-using-word2vec/`
- word2vec in Python
    - Rahuljha, *Word2Vec Implementation*,
      `https://towardsdatascience.com/a-word2vec-implementation-using-numpy-and-python-d256cf0e5f28`
    - Ivan Chen, *Word2vec from Scratch with NumPy*,
      `https://towardsdatascience.com/word2vec-from-scratch-with-numpy-8786ddd49e72`

Don't read this section

CBOW - !!! note to be used in a future !!!

The idea: training cycle

To be able to calculate weights updates we have to define an error function and then calculate required derivatives.

Our **goal is to maximize the likelihood of the context words given the center word**, i.e. we will calculate the probability of our model predicting the context words w_{t+c} given the center word w_t and **we will try to maximize that probability**. This likelihood can be represented using this formula:

$$L(\theta) = \prod_{c \in C} P(w_{t+c} | w_t; \theta)$$

where

$$\theta = \{W, W'\}$$

is a **set of weights** and

$$P(w_{t+c} | w_t; \theta) = \frac{e^{u_{c,t^*}}}{\sum_{i=1}^{N} e^{u_{c,i}}}$$

is a **conditional probabilities of the target word w_t, given the context word w_{t+c}**, where t^* is an index of word w_t in your one-hot vector representation, and N is a total number of all different words (the size of our vocabulary).

To put this equation in a form such that it is easy to take derivatives and make it a **minimization problem**, we will just take the log of the equation and multiply it by -1. Note that now multiplication will turn to summation as we have taken the log

$$J(\theta) = -\log L(\theta) = -\sum_{c \in C} \log P(w_{t+c} | w_t; \theta)$$