

Word Embedding : NPLM, Word2Vec, FastText

한국어 임베딩 4장 : 단어 수준 임베딩

Presented by | Jeong Minsu

2019.11.05.

A Neural Probabilistic Language Model – Bengio et al., 2003

기존 단어 모델의 한계

- 1) 학습 데이터에 존재하지 않는 n -gram이 포함된 문장이 나타날 확률 값을 0으로 부여
: back-off나 smoothing으로 일부 보완할 수 있지만 완전하지 않다.
- 2) 문장의 장기 의존성(long-term dependency)을 포착하기 어렵다.
: n -gram에서 n 이 커질수록 그 등장 확률이 0인 단어 시퀀스가 기하급수적으로 늘어난다.
- 3) 단어/문장 간 유사도를 계산할 수 없다.

NPLM 기본 구조

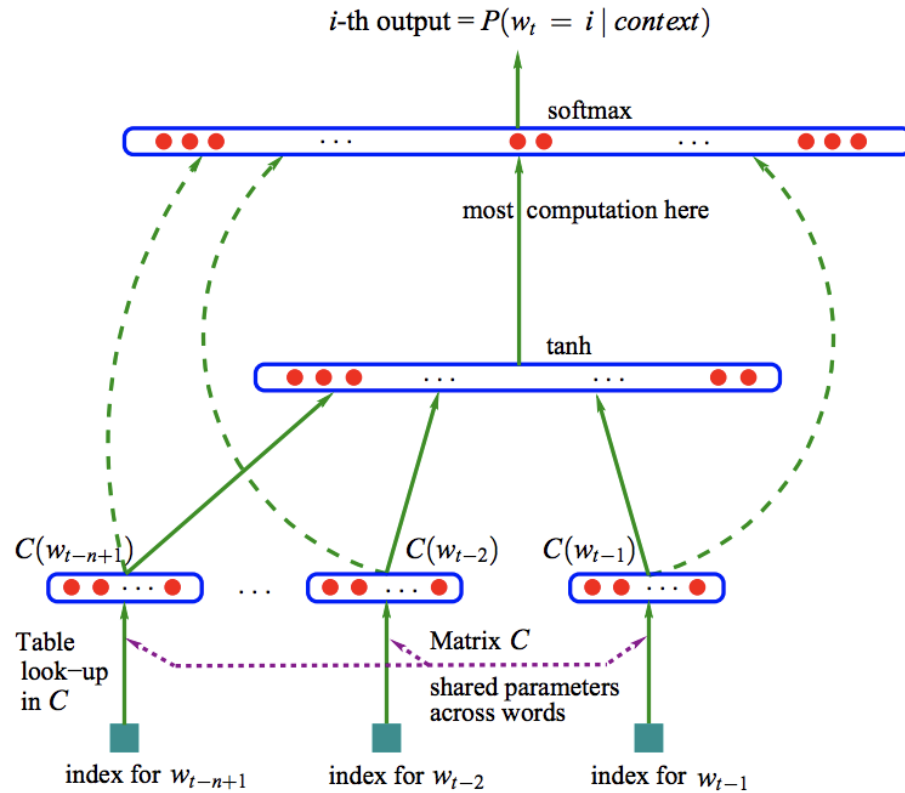


Figure 1: Neural architecture: $f(i, w_{t-1}, \dots, w_{t-n+1}) = g(i, C(w_{t-1}), \dots, C(w_{t-n+1}))$ where g is the neural network and $C(i)$ is the i -th word feature vector.

기본 아이디어:

직전까지 등장한 $n-1$ 개 단어들로 다음 단어를 맞추는 n -gram 언어 모델

“발 없는 말이 천리 간다”

↙ 4-gram

발, 없는, 말이, ??($t=4$)

없는, 말이, 천리, ??($t=5$)

NPLM 기본 구조

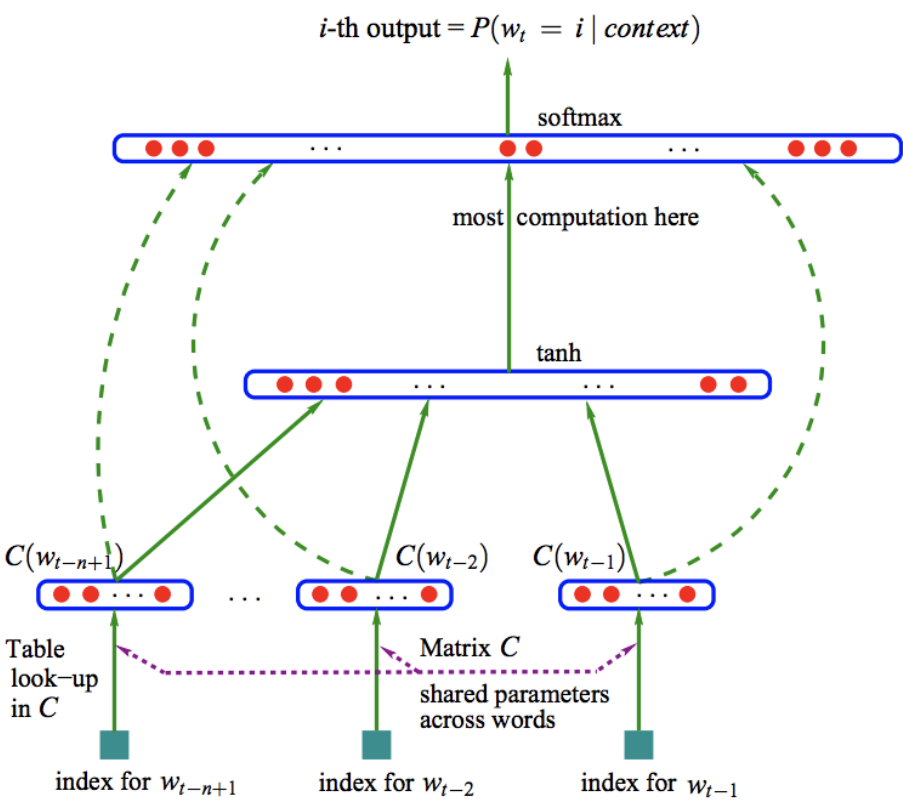


Figure 1: Neural architecture: $f(i, w_{t-1}, \dots, w_{t-n+1}) = g(i, C(w_{t-1}), \dots, C(w_{t-n+1}))$ where g is the neural network and $C(i)$ is the i -th word feature vector.

1. 행렬 C의 초기 값은 랜덤.

$$C = \begin{bmatrix} 11 & 18 & 25 \\ 10 & 12 & 19 \\ 4 & 6 & 13 \\ 23 & 5 & 7 \\ 17 & 24 & 1 \end{bmatrix} \begin{matrix} \text{발} \\ \text{없는} \\ \text{말} \\ \text{이} \\ \text{천리} \end{matrix}$$

2. 각 단어에 해당하는 원-핫 벡터를 행렬 C와 내적 (Look-up table)

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} 11 & 18 & 25 \\ 10 & 12 & 19 \\ 4 & 6 & 13 \\ 23 & 5 & 7 \\ 17 & 24 & 1 \end{bmatrix} = \begin{bmatrix} 23 & 5 & 7 \end{bmatrix}$$

NPLM 기본 구조

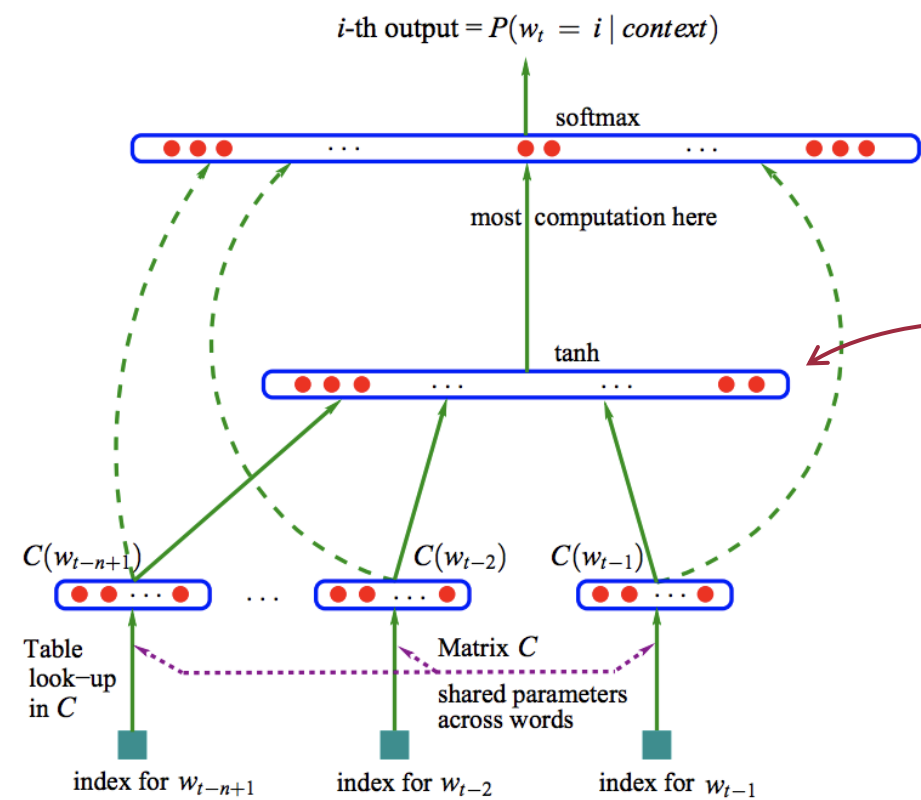


Figure 1: Neural architecture: $f(i, w_{t-1}, \dots, w_{t-n+1}) = g(i, C(w_{t-1}), \dots, C(w_{t-n+1}))$ where g is the neural network and $C(i)$ is the i -th word feature vector.

3. 단어들의 벡터를 붙임(concatenate).

$$x = [x_{t-1}, x_{t-2}, \dots, x_{t-n+1}]$$

$$x = [10, 12, 19, 4, 6, 13, 23, 5, 7]$$

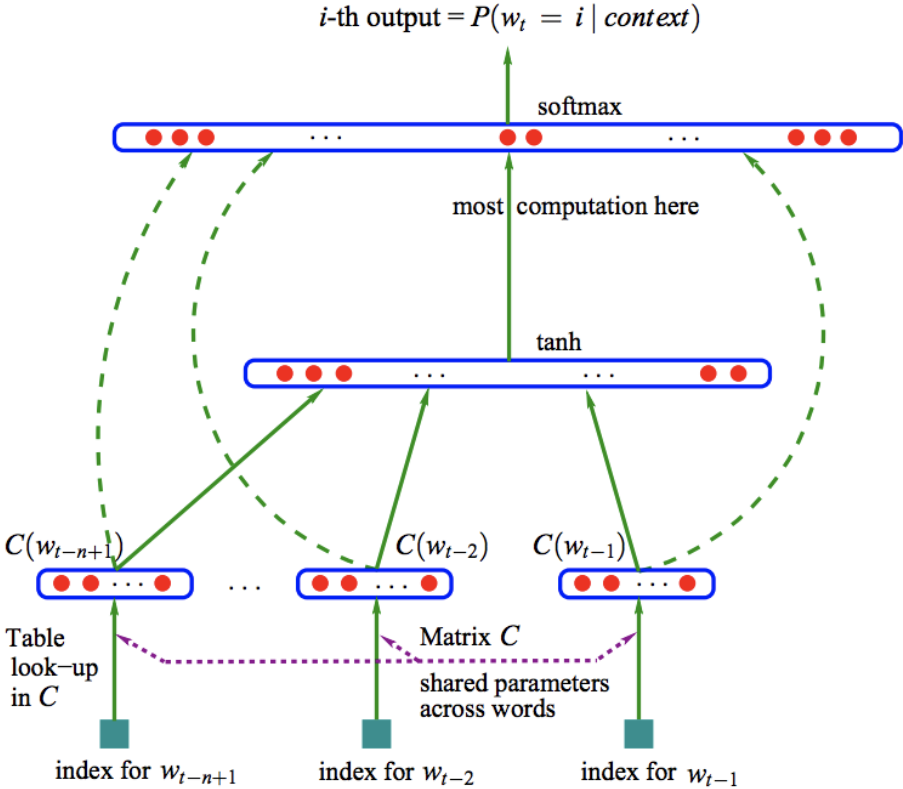
없는 말 이

4. 위에서 계산된 x가 모델의 입력으로 input layer, hidden layer, output layer를 거쳐 스코어 벡터 계산

$$y_{w_t} = b + Wx + U \tanh(d + Hx)$$

NPLM 기본 구조

5. 계산된 스코어 벡터(y)에 소프트맥스 함수를 적용하여 원래 단어의 인덱스와 비교해 loss를 계산하여 역전파



$$y_{w_t} = b + Wx + Utanh(d + Hx)$$

↓ softmax

$$P(w_t|w_{t-1}, \dots, w_{t-n+1}) = \frac{exp(y_{w_t})}{\sum_i exp(y_t)}$$

Figure 1: Neural architecture: $f(i, w_{t-1}, \dots, w_{t-n+1}) = g(i, C(w_{t-1}), \dots, C(w_{t-n+1}))$ where g is the neural network and $C(i)$ is the i -th word feature vector.

NPLM과 의미 정보

The cat is walking in the bedroom.

A dog was running in a room.

The cat is running in a room.

A dog is walking in a bedroom.

The dog was walking in the room.

$n=4$ 일 때,

walking을 예측하려면

NPLM과 의미 정보

The cat is walking in the bedroom.

A dog was running in a room.

The cat is running in a room.

A dog is walking in a bedroom.

The dog was walking in the room.

n=4 일 때,

walking을 예측하려면

NPLM과 의미 정보

The cat is walking in the bedroom.

A dog was running in a room.

The cat is running in a room.

A dog is walking in a bedroom.

The dog was walking in the room.

n=4 일 때,
walking을 예측하려면

The, cat, is, A, dog, was



0.1	0.2	0.1
0.2	0.1	0
⋮		
0.2	0.1	0.1
0.4	0.6	0.1

NPLM과 의미 정보

The cat is walking in the bedroom.

A dog was running in a room.

The cat is running in a room.

A dog is walking in a bedroom.

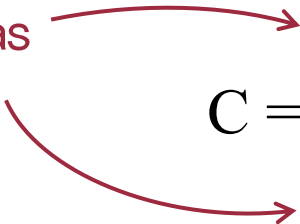
The dog was walking in the room.

n=4 일 때,

walking을 예측하려면

The, cat, is, A, dog, was

C =


$$\begin{bmatrix} 0.1 & 0.2 & 0.1 \\ 0.2 & 0.1 & 0 \\ \vdots & \vdots & \vdots \\ 0.2 & 0.1 & 0.1 \\ 0.4 & 0.6 & 0.1 \end{bmatrix}$$

walking을 맞추는 과정에서

발생한 손실(train loss)을 최소화하는

그래디언트(gradient)를 받아

동일하게 업데이트

: 벡터 공간에서 같은 방향으로 움직임!

NPLM과 의미 정보

- 기존 n-gram 모델에서는 학습 데이터에 한 번도 등장하지 않은 패턴에 대해서는 등장 확률을 0으로 부여하지만, NPLM에서는 문맥이 비슷한 다른 문장을 참고해 확률을 부여한다.
- 하지만, 학습 파라미터가 너무 많아 계산이 복잡하고, 모델의 과적합 가능성이 높다.

※NPLM 모델의 학습 파라미터

$$H \in R^{h \times (n-1)m}, \quad x_t \in R^{(n-1) \times m}, \quad d \in R^{h \times 1} \\ U \in R^{|V| \times h}, \quad b \in R^{|V|}, \quad y \in R^{|V|}, \quad C \in R^{m \times |V|}$$

Word2Vec from Google

「Efficient Estimation of Word Representations in Vector Space」, Mikolov et al., 2013a

: Skip-Gram와 CBOW(Continuous Bag of Words) 모델 제안.

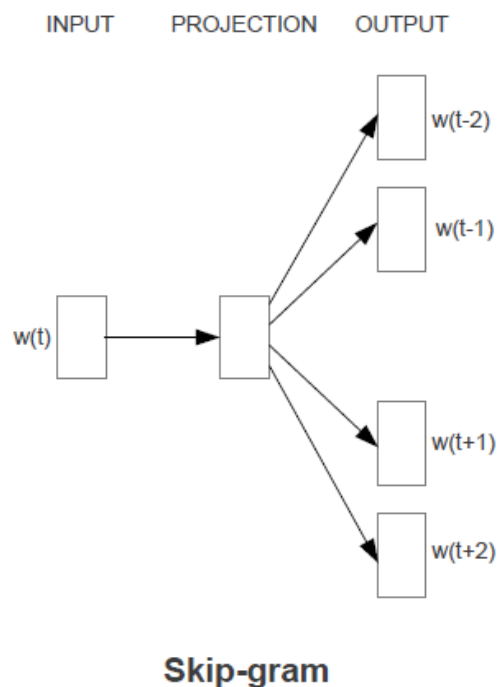
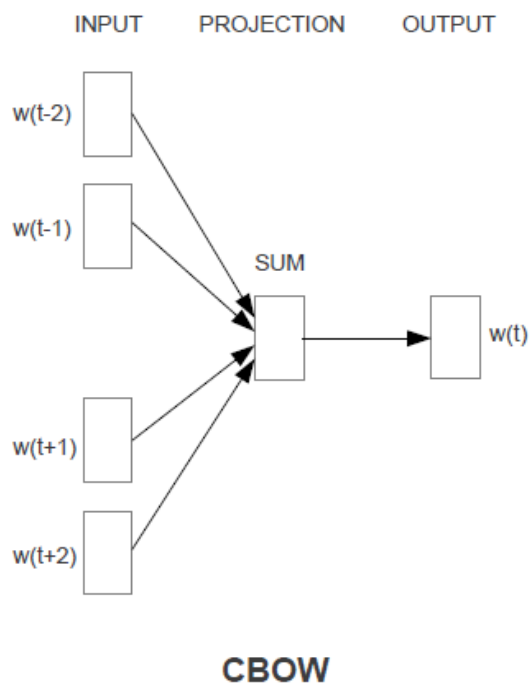
「Distributed Representations of Words and Phrase and their Compositionality」, Mikolov et al., 2013b

: 네거티브 샘플링(negative sampling), 서브 샘플링(sub-sampling) 등 최적화 기법 제안.

- 분포 가정을 기반으로 한 워드 임베딩 방법론.
distributional hypothesis :
비슷한 맥락에 등장하는 단어들은 유사한 의미를 지니는 경향이 있다.
- window 내에 등장하지 않으면 결과값을 줄이고, 등장할 경우 결과값을 키우기 때문에
단어-문맥행렬처럼 count 기반 방법론과 같이 자주 같이 등장하는 단어들의 정보(co-occurrence)를
보존하기 때문에 Word2Vec은 본질적으로 기존 count 기반의 방법론과 다르지 않다.

Word2Vec

Word2Vec의 기본 구조



CBOW (Continuous Bag of Words) :

주변 단어(context word)를 가지고
중심 단어(target word)를 예측

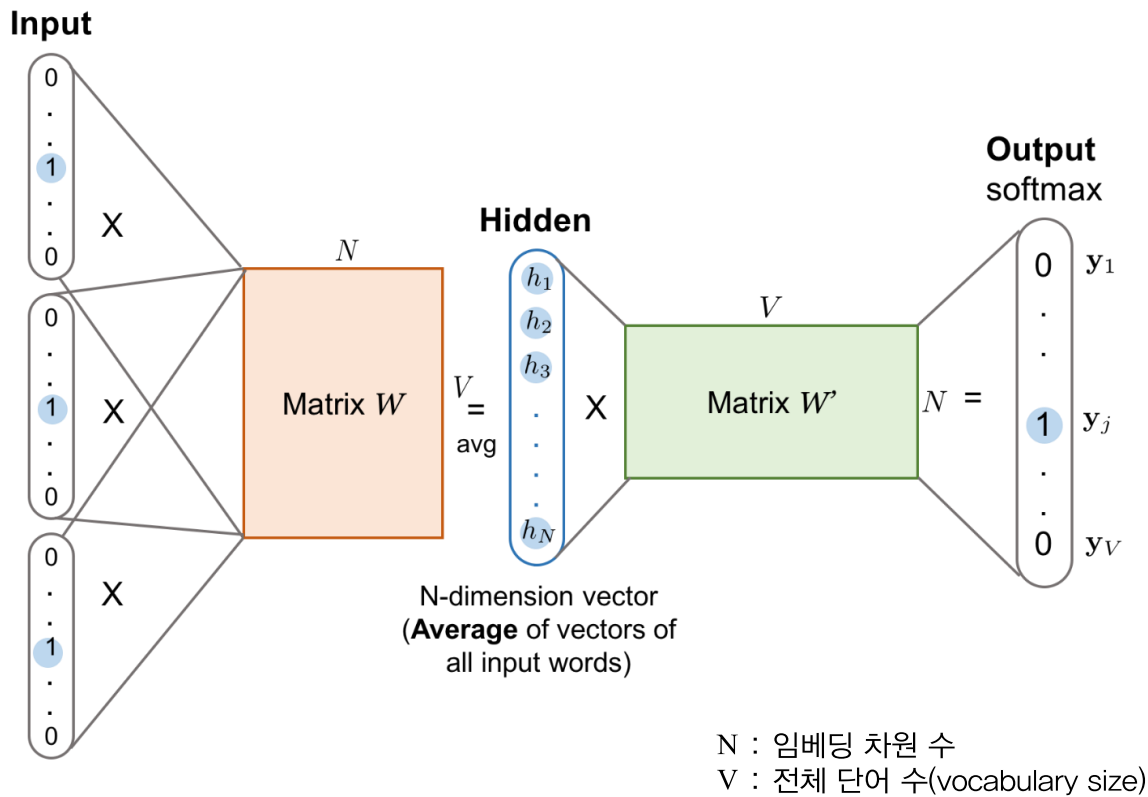
Skip-gram :

중심 단어(target word)를 가지고
주변 단어(context word)를 예측

CBOW보다 학습 데이터량이 많고, 주변 단어끼리 독립적이라는 가정 때문에 더 많이 쓰임.

Figure 1: New model architectures. The CBOW architecture predicts the current word based on the context, and the Skip-gram predicts surrounding words given the current word.

CBOW 모델 구조

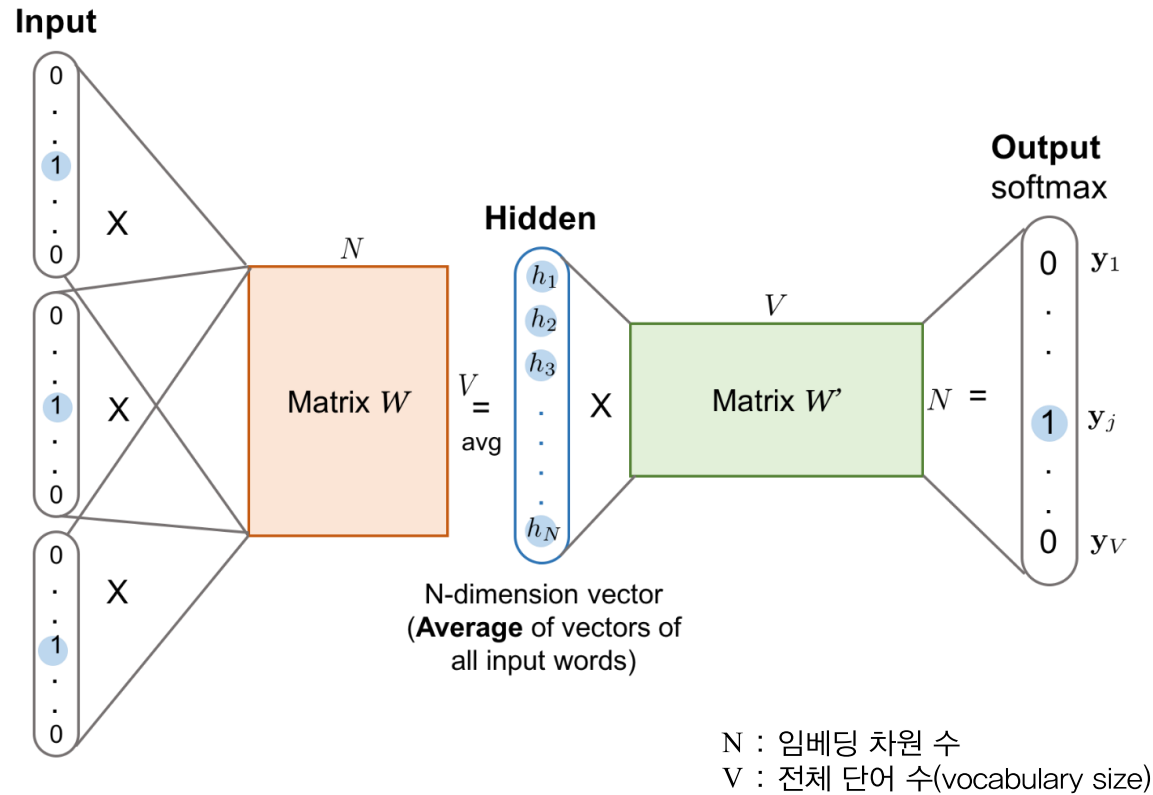


1. 문맥 단어들(context words)의 원-핫 벡터(one-hot vector)와 행렬 W (Input Layer Hidden Layer로 가는 파라미터 행렬)이 곱해져서(look-up) 평균을 냄.

$$\begin{bmatrix} 0 & 0 & 0 & 1 & 0 \end{bmatrix} \times \begin{bmatrix} 11 & 18 & 25 \\ 10 & 12 & 19 \\ 4 & 6 & 13 \\ 23 & 5 & 7 \\ 17 & 24 & 1 \end{bmatrix} = \begin{bmatrix} 23 & 5 & 7 \end{bmatrix}$$

$2 \times w$ 개(window size)의 벡터들의 평균(N 의 크기)이 Hidden Layer가 됨.

CBOW 모델 구조



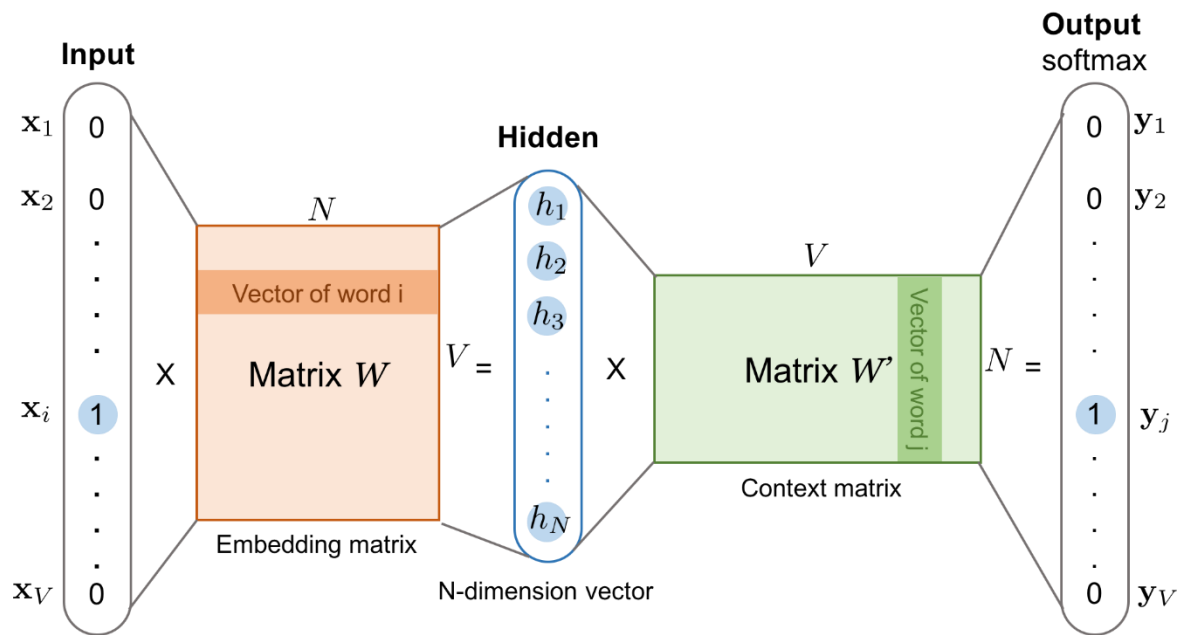
2. 앞에서 구한 Hidden Layer 벡터 값에 행렬 W' 을 곱해서 각 단어에 대한 score를 만듦.

3. softmax로 확률 값 계산.

4. Loss function으로 Loss 계산 후, 역전파(backpropagation)

$$\begin{aligned}
 \text{minimize } J &= -\log P(w_c | w_{c-m}, \dots, w_{c+m}) \\
 &= -\log P(u_c | v) \\
 &= -\log \frac{\exp(u_c^T \hat{v})}{\sum_{j=1}^{|V|} \exp(u_j^T \hat{v})} \\
 &= -u_c^{\text{intercal}} \hat{v} + \log \sum_{j=1}^{|V|} \exp(u_j^T \hat{v})
 \end{aligned}$$

Skip-gram 모델 구조



N : 임베딩 차원 수
 V : 전체 단어 수(vocabulary size)

1. (target, context) 쌍의 데이터 준비

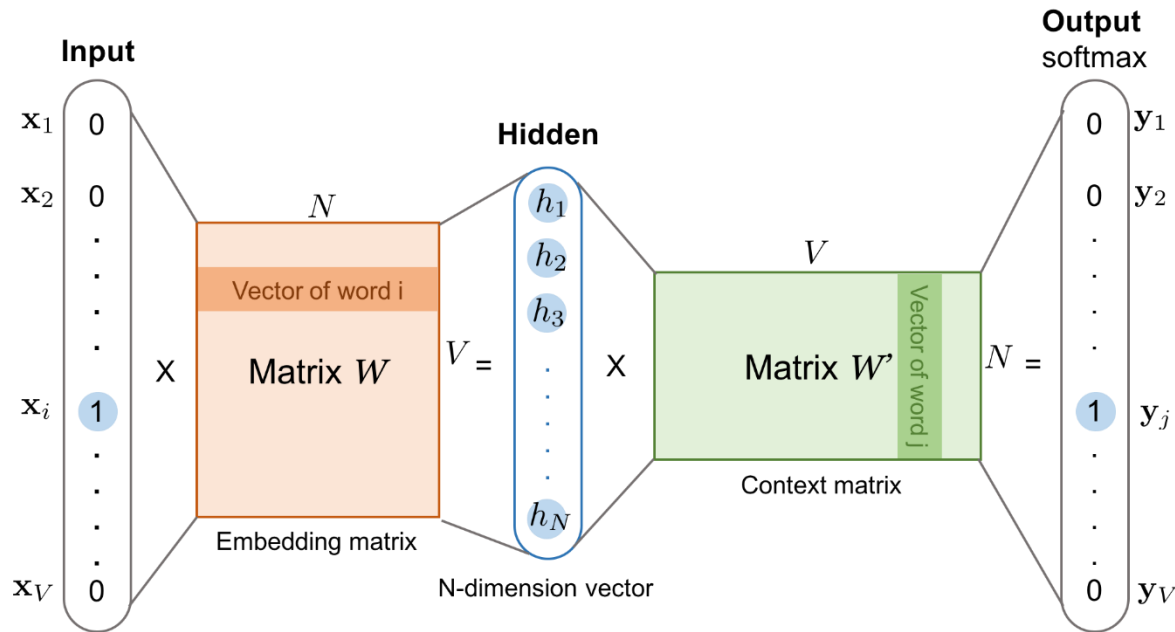
“The mouse is **running** in a room.”

$w = 2$

(“running”, “mouse”)
(“running”, “is”)
(“running”, “in”)
(“running”, “a”)

2. 타깃 단어(target word)를 원-핫 벡터로 만들고, 행렬 W 와 곱해서 embedded vector를 구함.

Skip-gram 모델 구조



N : 임베딩 차원 수
 V : 전체 단어 수(vocabulary size)

3. embedded vector를 두 번째 파라미터 행렬 W' 와 곱해서 score vector로 만들.

4. 각 score vector를 softmax 함수 통해 확률 값 계산.

5. Loss function으로 Loss 계산 후, 역전파(backpropagation)

$$\begin{aligned}
 \text{minimize } J &= -\log P(w_{c-m}, \dots, w_{c-1}, w_{c+1}, \dots, w_{c+m} | w_c) \\
 &= -\log \prod_{j=0, j \neq m}^{2m} P(w_{c-m+j} | w_c) \\
 &= -\log \prod_{j=0, j \neq m}^{2m} \frac{\exp(u_{c-m+j}^T v_c)}{\sum_{k=1}^{|V|} \exp(u_k^T v_c)} \\
 &= -\sum_{j=0, j \neq m}^{2m} u_{c-m+j}^T v_c + 2m \log \sum_{k=1}^{|V|} \exp(u_k^T v_c)
 \end{aligned}$$

Word2Vec 모델 최적화 기법

※ NLPM의 모델 파라미터

$$H \in R^{h \times (n-1)m}, \quad x_t \in R^{(n-1) \times m}, \quad d \in R^{h \times 1}$$
$$U \in R^{|V| \times h}, \quad b \in R^{|V|}, \quad y \in R^{|V|}, \quad C \in R^{m \times |V|}$$



※ Word2Vec의 모델 파라미터

$$W \in R^{|V| \times N}, W' \in R^{N \times |V|}$$

※ Word2Vec 모델 계산량

1) CBOW 모델의 계산량

- 1) 2*w개의 단어를 Hidden Layer로 보내는 $2*w \times N$
- 2) Hidden Layer에서 Output Layer로 가는 $N \times V$

→ 전체 계산량 = $2*w \times N + N \times V$

2) Skip-gram 모델의 계산량

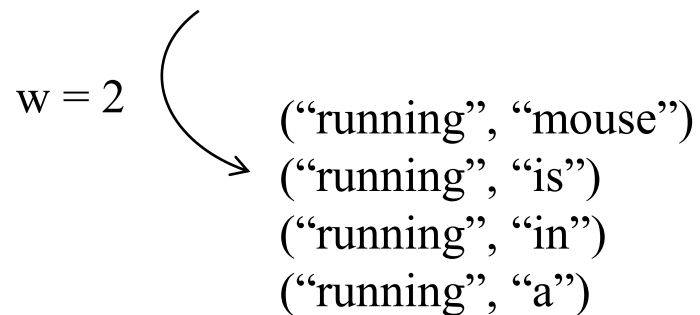
- 1) 타깃 단어를 Hidden Layer로 보내는 $1 \times N$
- 2) Hidden Layer에서 Output Layer로 가는 $N \times V$

데이터가 2*w 늘어남.

→ 전체 계산량 = $(1 \times N + N \times V) * 2*w$

Word2Vec 모델 최적화 기법 – (1) Subsampling Frequent words

“The mouse is **running** in a room.”



- “is”, “in”, “a” 와 같은 단어들은 전체 데이터 안에서 매우 자주 등장할 것이다.
- 빈도 수가 높은 단어들은 학습 횟수도 많음.
- 하지만, 학습 횟수에 비해 의미적으로 중요하지 않기 때문에, 모델의 정확도에 기여하는 부분이 적다.

Word2Vec 모델 최적화 기법 – (1) Subsampling Frequent words

$$P_{\text{subsampling}}(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}}$$

t : 하이퍼파라미터, 0.00001 값 추천.

f() : 전체 데이터 중 단어의 등장 빈도 수

- 전체 100번 중 한 번 나타나는 꼴의 $f=0.01$ 인 단어는 위 식으로 계산한 $P(w)$ 가 0.9684가 되기 때문에 해당 단어가 가질 수 있는 100번의 학습 기회 가운데 3~4번 학습에서 제외시킴.



Input:


$$1 - \sqrt{\frac{0.00001}{0.01}}$$

Result:

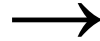
0.9683772...

Word2Vec 모델 최적화 기법 – (2) Negative Sampling

“The mouse is **running** in a room.”

$w = 2$ 

- (“running”, “mouse”)
- (“running”, “is”)
- (“running”, “in”)
- (“running”, “a”)



Positive sample

- (“running”, “mouse”)
- (“running”, “is”)
- (“running”, “in”)
- (“running”, “a”)

Negative sample

- (“running”, “natural”)
- (“running”, “language”)
- (“running”, “process”)
- (“running”, “hard”)
- ...

- 기존의 확률 계산(=전체 어휘 집합의 크기만큼의 차원 수를 가진 softmax 계산)에서는 모든 단어에 대해서 전체 경우를 구함.
- 하지만, Negative Sampling에서는 현재 window에 등장하지 않는 단어를 특정 개수(k)만 뽑아서 확률 계산.

Word2Vec 모델 최적화 기법 – (2) Negative Sampling

※ 네거티브 샘플링 확률

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

$f()$: 전체 데이터 중 단어의 등장 빈도 수

$\frac{3}{4}$: 다른 값들에 비해 성능을 가장 잘 내는 고정 값

- 빈도 수에 $\frac{3}{4}$ 승을 해줬기 때문에, 빈도가 높은 값은 살짝 낮게, 빈도가 낮은 값은 높게 보정(?)하는 역할을 함.
- 해당 쌍이 포지티브 샘플(+)인지, 네거티브 샘플(-)인지 이진 분류(binary classification)하는 과정을 거침.
(=매 스텝마다 차원 수가 2인 시그모이드를 k+1회 계산)

Word2Vec 모델 최적화 기법 - (2) Negative Sampling

※ t, c가 포지티브 샘플(=t 주변에 c가 존재)일 확률

$$P(+|t, c) = \frac{1}{1 + \exp(-u_t v_c)}$$

u_t: 타겟 단어(t)에 해당하는 워드 벡터
v_c: 주변 단어(c)에 해당하는 워드 벡터

- 실제 t, c가 포지티브 샘플이라면 위 식을 최대화하는 방향으로 학습해야 함.

$$P(+|t, c) = \frac{1}{1 + \exp(-u_t v_c)}$$

최소가 되어 함. $u_t v_c$: 두 벡터의 내적은 벡터 공간의 코사인 유사도(cosine similarity)와 비례.

Word2Vec 모델 최적화 기법 - (2) Negative Sampling

※ t, c가 포지티브 샘플(=t 주변에 c가 존재)일 확률

$$P(+|t, c) = \frac{1}{1 + \exp(-u_t v_c)}$$

u_t: 타겟 단어(t)에 해당하는 워드 벡터
v_c: 주변 단어(c)에 해당하는 워드 벡터

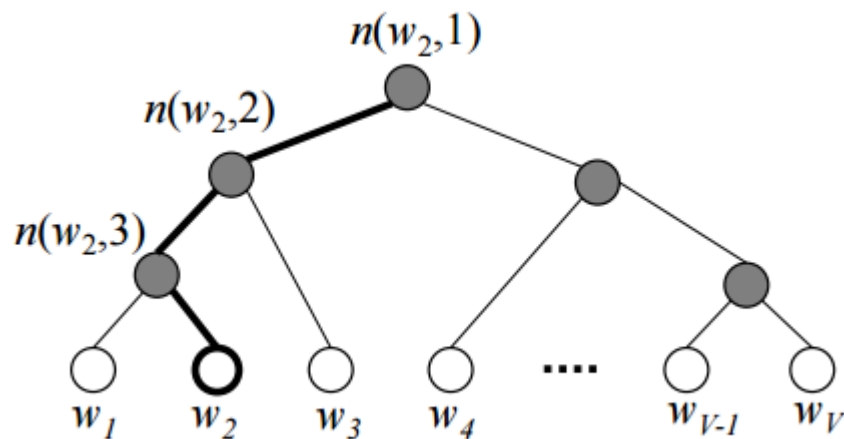
- 실제 t, c가 포지티브 샘플이라면 위 식을 최대화하는 방향으로 학습해야 함.

$$P(+|t, c) = \frac{1}{1 + \exp(-u_t v_c)}$$

최대가 되어야 함.

$u_t v_c$: 두 벡터의 내적은 벡터 공간의 코사인 유사도(cosine similarity)와 비례.

Word2Vec 모델 최적화 기법 – (3) Hierarchical Softmax



$n(w, i)$: 트리의 root에서 단어 leaf w 까지 길에 놓여있는 i 번째 노드.
 ※ leaf에 word vector가 포함되어 있음.

- 기존의 확률 계산은 전체 어휘 집합의 크기만큼의 차원 수를 가진 softmax를 계산 함.
- 하지만 Hierarchical Softmax는 Tree구조로 계층적으로 Softmax를 계산한다.
- Hierarchical Softmax와 Negative Sampling은 확률 계산량을 줄이기 위한 방법으로 목적이 같음. 택일해서 사용.
- Hierarchical Softmax는 빈번하게 출현하지 않는 단어들에 유리하고, negative sampling은 빈번하게 출현하는 단어에 좀 더 적합하다고 함.

Word2Vec

Word2Vec from gensim

```
In [17]: corpus_fname = "../data/tokenized/korquad_mecab.txt"
```

```
In [18]: model_fname = "../data/word-embeddings/word2vec/word2vec"
```

```
In [19]: from gensim.models import Word2Vec
```

```
In [20]: corpus = [sent.strip().split(" ") for sent in open(corpus_fname, 'r').readlines()]
```

```
In [21]: model = Word2Vec(corpus, size=100, workers=8, sg=1)
```

```
In [22]: model.save(model_fname)
```

```
/usr/local/lib/python3.5/dist-packages/smart_open/smart_open_lib.py:398: UserWarning: This function is deprecated, use smart_open.open instead. See the migration notes for details: https://github.com/RaRe-Technologies/smart\_open/blob/master/README.rst#migrating-to-the-new-open-function  
'See the migration notes for details: %s' % _MIGRATION_NOTES_URL
```

```
In [30]: model_result = model.wv.most_similar("대통령")
```

```
In [31]: model_result
```

```
Out [31]: [('문재인', 0.7707852125167847),  
           ('클린턴', 0.7434890270233154),  
           ('박근혜', 0.740337610244751),  
           ('노무현', 0.740064263343811),  
           ('루즈벨트', 0.7400109171867371),  
           ('당선자', 0.7335081100463867),  
           ('비서', 0.7282658219337463),  
           ('아이젠하워', 0.7249844074249268),  
           ('부통령', 0.721725344657898),  
           ('버락', 0.7215172052383423)]
```



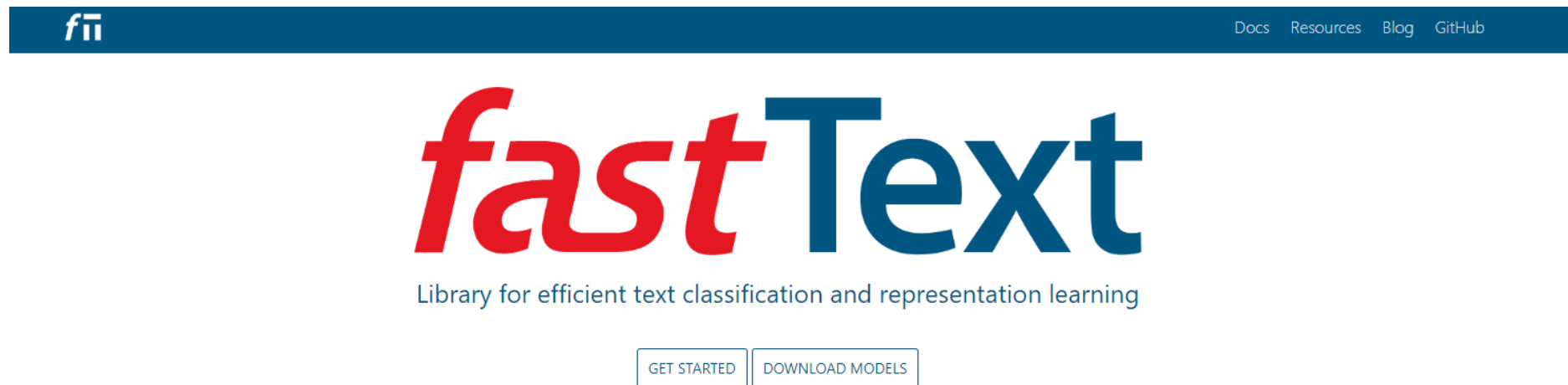
「Enriching Word Vectors with Subword Information」, Bonjanowski et al., 2016

: 개별 단어가 아닌 n-gram 단위의 character embedding 방법론.

「Advances in Pre-Training Distributed Word Representation」, Mikolov et al., 2017

: fastText 완성

<https://fasttext.cc/>



기존 언어 모델의 비판

- 단어의 형태학적 특성을 반영하지 못했다.

기존의 Word2Vec과 GloVe 등은 단어 단위의 개별적 임베딩을 하기 때문에 teach, teacher, teachers 등과 같이 실제 유사한 의미를 가짐에도 불구하고 임베딩 벡터가 유사하게 구성되지 않았다.

- 희소한 단어를 임베딩하기 어렵다.

기존의 Word2Vec은 Distribution hypothesis를 기반으로 학습하기 때문에, 출현 횟수가 많은 단어는 임베딩을 잘하지만, 출현 횟수가 적은 단어는 제대로 임베딩을 하지 못한다.

fastText 모델 구조

- 단어의 시작과 끝에 <, >를 추가해 단어 경계를 표현.
- 각 단어를 문자(character) 단위 n-gram으로 표현.
- 그리고 <원래 단어>를 추가.
- 아래와 같이 5개의 문자 단위 n-gram 벡터의 합으로 원래 단어의 임베딩을 표현.

“슈퍼마켓”

↙ tri-gram ※<슈퍼>는 다른 단어

<슈퍼, 슈퍼마, 퍼마켓, 마켓>, <슈퍼마켓>

$$\rightarrow u_{\text{슈퍼마켓}} = (z_{\langle \text{슈퍼} \rangle} + z_{\text{슈퍼마}} + z_{\text{퍼마켓}} + z_{\text{마켓}} + z_{\langle \text{슈퍼마켓} \rangle}) / 5$$

fastText 모델 구조

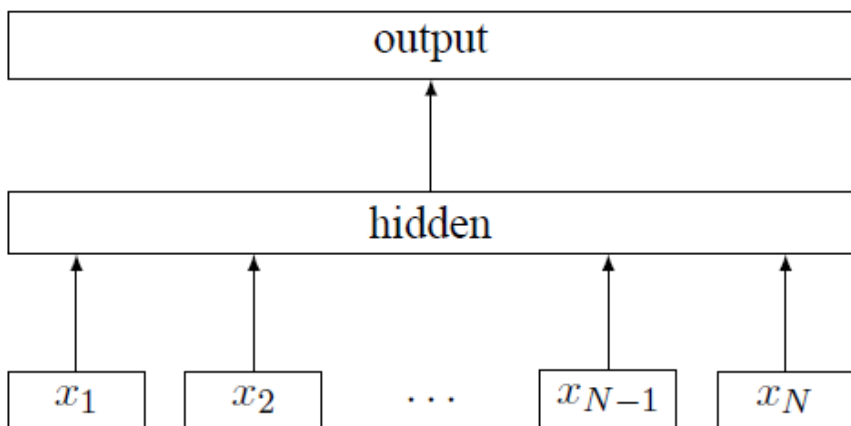


Figure 1: Model architecture of fastText for a sentence with N ngram features x_1, \dots, x_N . The features are embedded and averaged to form the hidden variable.

1. Look-up table을 이용해 단어의 임베딩을 구함.
2. 단어 벡터들을 평균을 Input으로 사용.
3. 나머지는 Skip-gram 모델과 같음.
4. Skip-gram과 마찬가지로 Negative Sampling을 사용.

$$P(+|t, c) = \frac{1}{1 + \exp(-\underbrace{u_t v_c}_{\text{최대가 되어야 함}})}$$

※ fastText의 로그우도 함수

$$L(\theta) = \underbrace{\log P(+|t_p, c_p)}_{\text{1개의 포지티브 샘플}} + \sum_{i=1}^k \underbrace{\log P(-|t_{n_i}, C_{n_i})}_{\text{k개의 네거티브 샘플}}$$

fastText 효과

- n-gram의 문자단위 임베딩으로 Out-of-Vocabulary(OOV)를 처리할 수 있다.
- 단어의 내부 구조를 반영할 수 있다. (BPE; Byte Pair Encoding 알고리즘과 유사)
- 새로운 단어가 등장해도 기존의 n-gram vector를 찾아서 summation하면 재학습 과정 없이 대응할 수 있음.
- 어휘의 구문적(syntactic) 변화 규칙을 잘 잡아낼 수 있다.
- 한글을 자소 단위로 분해하여 한국어 임베딩에서 효과가 높다.
- 연구진들이 fastText로 학습된 언어의 임베딩 벡터와 코드를 공개했음 !

fastText 실습

```
# models/fastText/fasttext skipgram -input data/tokenized/korquad_mecab.txt -output data/word-embeddings/fasttext/fasttext
Read 4M words
Number of words: 30749
Number of labels: 0
Progress: 100.0% words/sec/thread: 178909 lr: 0.000000 loss: 1.933963 ETA: 0h 0m
```

```
In [1]: ▶ import fasttext
```

```
In [15]: ▶ model = fasttext.load_model('./data/word-embeddings/fasttext/fasttext.bin')
```

```
In [18]: ▶ import models.word_eval
```

```
In [20]: ▶ model = models.word_eval.WordEmbeddingEvaluator(
    vecs_txt_fname = './data/word-embeddings/fasttext/fasttext.vec',
    vecs_bin_fname = './data/word-embeddings/fasttext/fasttext.bin',
    method='fasttext', dim=100, tokenizer_name='mecab')
```


fastText 실습

〈Word2Vec〉

```
In [30]: model_result=model.wv.most_similar("대통령")
```

```
In [31]: model_result
```

```
Out [31]: [('문재인', 0.7707852125167847),
            ('클린턴', 0.7434890270233154),
            ('박근혜', 0.740337610244751),
            ('노무현', 0.740064263343811),
            ('루즈벨트', 0.7400109171867371),
            ('당선자', 0.7335081100463867),
            ('비서', 0.7282658219337463),
            ('아이젠하워', 0.7249844074249268),
            ('부통령', 0.721725344657898),
            ('벼락', 0.7215172052383423)]
```

〈fasttext〉

```
In [21]: ▶ model.most_similar("대통령")
```

```
Out [21]: [('김대통령', 0.8329841395601176),
            ('루스벨트', 0.7437526865255558),
            ('대통령령', 0.7407945029013183),
            ('김영삼', 0.7265201743690457),
            ('부통령', 0.7254299801651853),
            ('문재', 0.7246341468773319),
            ('대통', 0.7164657819064795),
            ('시어도어', 0.7137464581236292),
            ('메드베데프', 0.7131546265527343),
            ('아이젠하워', 0.711914117801366)]
```

fastText 실습

〈Word2Vec〉

```
In [32]: model.wv.most_similar("하였다")
```

```
-----
KeyError                                Traceback (most recent call
<ipython-input-32-701a3d6c4b7f> in <module>
----> 1 model.wv.most_similar("하였다")

/usr/local/lib/python3.5/dist-packages/gensim/models/keyedvectors.py i
indexer)
    550             mean.append(weight * word)
    551         else:
--> 552             mean.append(weight * self.word_vec(word, use_n
    553                 if word in self.vocab:
    554                     all_words.add(self.vocab[word].index)

/usr/local/lib/python3.5/dist-packages/gensim/models/keyedvectors.py i
    465             return result
    466         else:
--> 467             raise KeyError("word '%s' not in vocabulary" % wor
    468
    469     def get_vector(self, word):
```

```
KeyError: "word '하였다' not in vocabulary"
```

〈fasttext〉

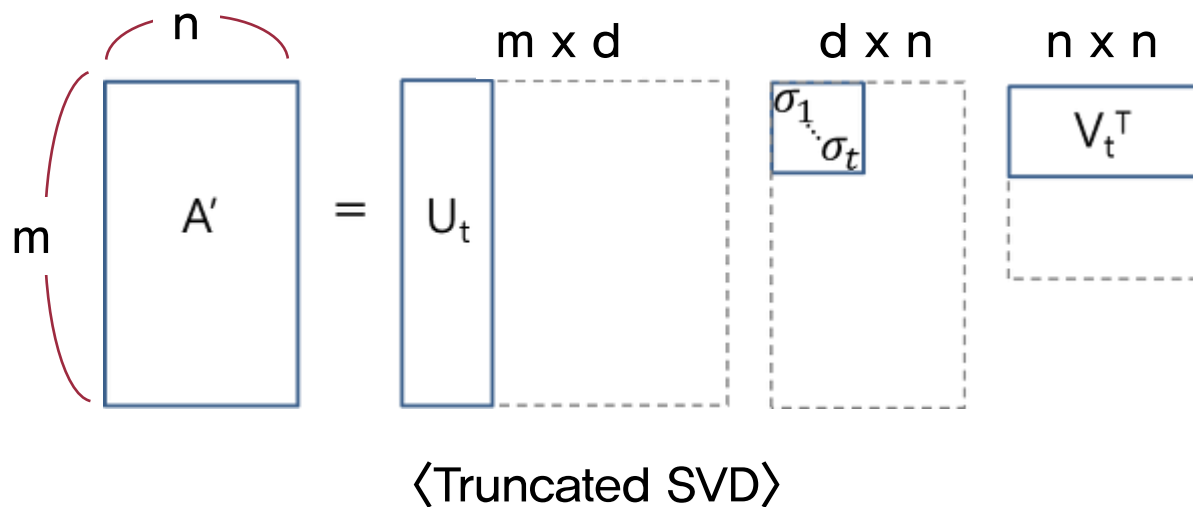
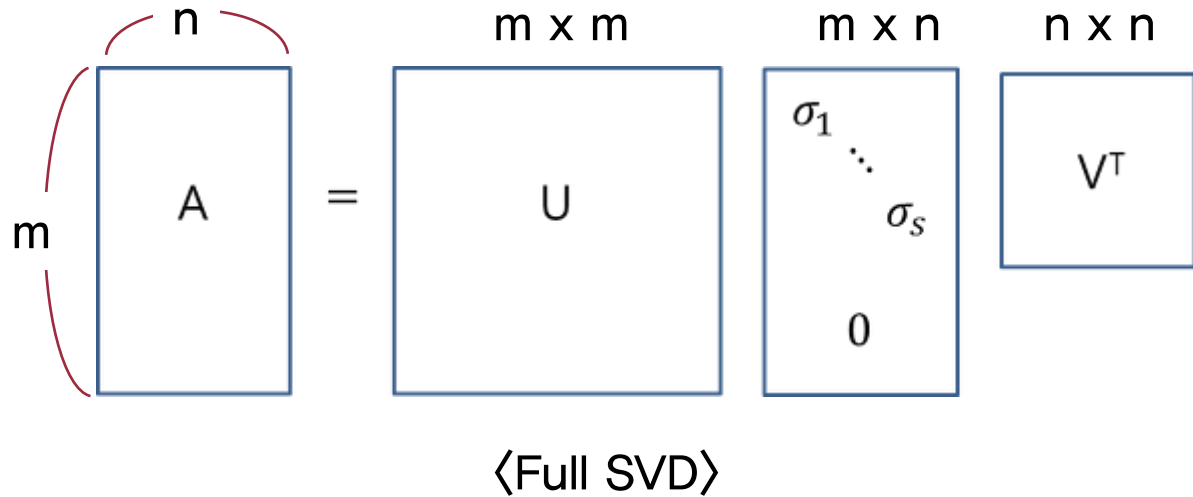
```
[22]: ► model.most_similar("하였다")
```

```
Out [22]: [('였 ', 0.8904696920920094),
            ('했 ', 0.8605284020073041),
            ('다 ', 0.8308236441002503),
            ('.', 0.7954372016535329),
            ('고 ', 0.7622287356569821),
            ('또한 ', 0.7527888891194932),
            ('는 ', 0.748023352482071),
            ('을 ', 0.7463404381964546),
            ('도 ', 0.7356440465421861),
            ('굳히 ', 0.7322947669529871)]
```

LSA, Latent Semantic Analysis ; 잠재 의미 분석

- 토픽 모델링을 위해 최적화된 알고리즘은 아니지만, 토픽 모델링이라는 분야에 아이디어를 제공한 알고리즘.
- BoW에 기반한 DTM이나 TF-IDF는 기본적으로 단어의 빈도 수를 이용한 수치화 방법이기 때문에 단어의 의미를 고려하지 못한다는 단점이 있었음.
- Matrix factorization 기반의 방법.
- DTM의 잠재된(Latent) 의미를 이끌어내는 방법.

특이값 분해 (Singular Value Decomposition)



1. 기존 SVD(Full SVD)는 행렬을 분해하는 기법.
2. Truncated SVD는 SVD를 통해 얻어진 Σ 행렬에서 대각 성분이 0으로 구성된 부분과 singular value까지 제거하여 얻을 수 있음.
3. 차원이 축소되면서 데이터의 (1) 노이즈(해당 차원에서 필요하지 않은 정보)와 (2) 희소성(sparsity) 이 제거된다.
4. A 가 단어 \times 문서 행렬이라면, 단어와 문서 간 내재적 의미를 효과적으로 보존할 수 있다.

reference

<https://reniew.github.io/21/>

<https://lilianweng.github.io/lil-log/2017/10/15/learning-word-embedding.html>

<https://dalpo0814.tistory.com/7>

https://inspiringpeople.github.io/data%20analysis/word_embedding/

<https://wikidocs.net/24949>