NLP 스터디 2주차

한국어 임베딩

4장 단어 수준 임베딩

4.1 NPLM

4.2 Word2Vec

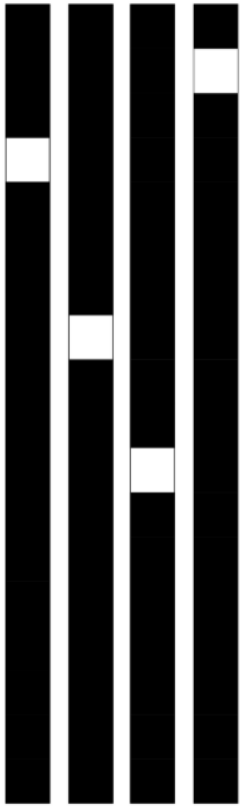
4.3 FastText

이영현

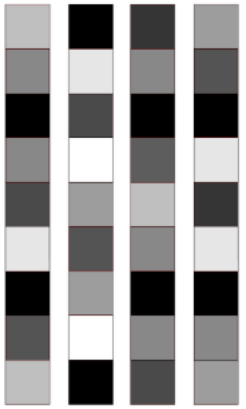
시작하기 전에, 임베딩은…

- 단어와 벡터를 연관 짓는 방법
  단어 사이에 있는 의미 관계를 반영해야 함

- 저차원의 실수형 벡터
  (희소, 고차원, 수동 인코딩 vs
  밀집, 저차원, 데이터로부터 학습)



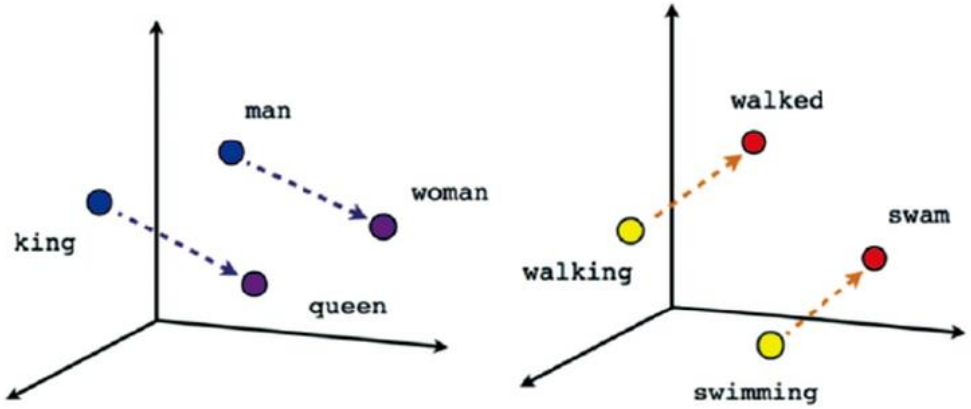
One-hot word vectors:
- Sparse
- High-dimensional
- Hard-coded



Word embeddings:
- Dense
- Lower-dimensional
- Learned from data

단어 임베딩

- 언어를 기하학적 공간에 매핑하는 것



Male-Female　　Verb tense

간단한 예시)

Man벡터 에 female 벡터를 더하면 woman 벡터가 됨

현재형에서 과거형으로 이동하는 것을 같은 벡터로 나타낼 수 있음

단어 임베딩 공간은 전형적으로 이런 해석이 가능하고 잠재적으로 유용한 수천 개의 벡터를 특성으로 가짐

단어 임베딩 in keras

단어 인덱스 -> Embedding 층 -> 연관된 단어 벡터

- Embedding층의 객체를 생성할 때 가중치는 다른 층과 마찬가지로 랜덤하게 초기화
- 훈련하면서 이 단어 벡터는 역전파를 통해 점차 조정되어 이어지는 모델이 사용할 수 있도록 임베딩 공간을 구성함
- 훈련이 끝나면 임베딩 공간은 특정 문제에 특화된 구조를 많이 가지게 됨

```
from keras.models import Sequential
from keras.layers import Embedding, Flatten, Dense

model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=maxlen))
model.add(Flatten())
model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.summary()
```

NPLM

기존 언어 모델의 단점Bengio et al.(2003)

- 학습 데이터에 존재하지 않는 n-gram이 포함된 문장이 나타날 확률 값을 0으로 부여
  -〉 백오프(n개보다 작은 빈도)나 스무딩(+k)으로 보완 가능하지만 완전한 것은 아님
- N-gram 모델의 n을 5이상으로 길게 설정할 수 없음
  -〉 등장 확률이 0인 단어 시퀀스가 기하급수적으로 늚
  trade-off: n이 너무 작으면 근사의 정확도는 현실의 확률분포와 멀어짐
- 단어/문장 간 유사도를 계산할 수 없음

NPLM: 기존 언어 모델의 한계를 일부 극복한 언어 모델, 자체가 단어 임베딩 역할 수행 가능

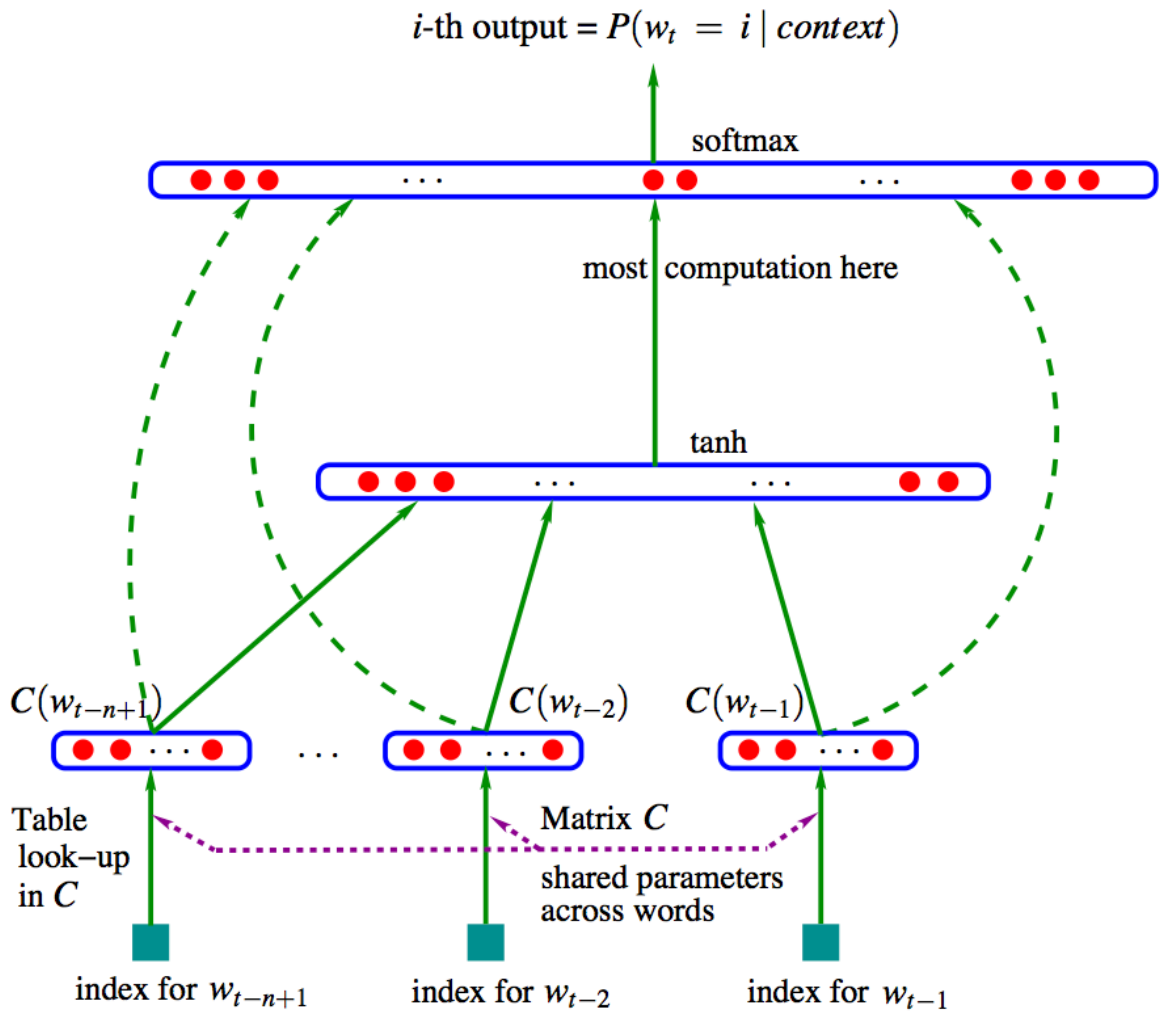
NPLM

단어 시퀀스가 주어졌을 때 다음 단어가 무엇인지 맞추는 과정에서 학습

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

발, 없는, 말

천리, 간다



NPLM

NPLM의 출력

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

P(천리|발, 없는, 말이)
P(간다|없는, 말이, 천리)
를 각각 높인다는 의미

w_1 = 발, w_2 = 없는, w_3 = 말이 -〉 w_4 = 천리를 잘 예측한다는 의미
(n=4, t=4)

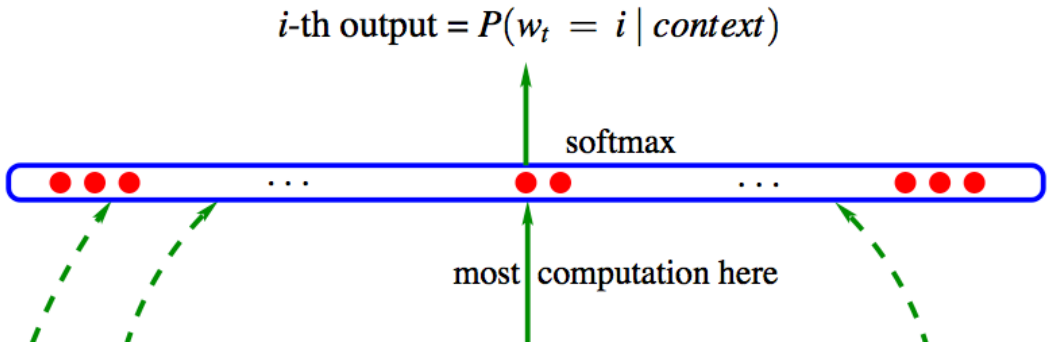
NPLM

NPLM 말단의 출력

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

= |V|차원의 스코어 벡터 y_{w_t}에 소프트맥스 함수를 적용한 |V|차원의 확률 벡터

NPLM은 확률 벡터에서 가장 높은 요소의 인덱스에 해당하는 단어가 실제 정답 단어와 일치하도록 학습



NPLM

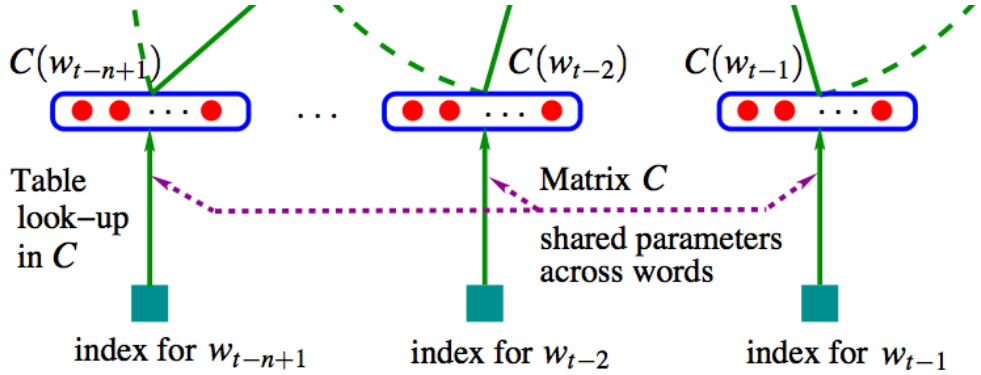
NPLM의 입력

문장 내 t번째 단어(w_t)에 대응하는 단어 벡터 x_t를 만드는 과정

$$x_t = C(w_t)$$

* $|V| \times m$ 크기를 갖는 커다란 행렬 C에서 w_t에 해당하는 벡터를 참조한 형태

(|V|: 어휘집합 크기, m: x_t의 차원 수)



NPLM

참조

$$C(w_t) = [0\ \ 0\ \ 1] \times \begin{bmatrix} 11 & 18 & 25 \\ 10 & 12 & 19 \\ 23 & 5 & 7 \end{bmatrix} = [23\ \ 5\ \ 7]$$

$C(w_t)$는 행렬 C와 w_t에 해당하는 원-핫벡터를 내적inner product한 것과 같음

= C라는 행렬에서 w_t에 해당하는 행만 참조하는 것과 동일

NPLM 의 입력 벡터

없는, 말이, 천리 로 간다 예측

세 개 단어에 해당하는 열 벡터를 C에서 참조한 뒤 concatenate 하면

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

n(고려 대상 n-gram 개수): 4, t=5

발 없는 말이 천리 간다

NPLM

학습 종료되면 행렬 C를 각 단어에 해당하는 m차원 임베딩으로 사용

아래식에 소프트맥스 함수 적용

정답 단어인 간다의 인덱스와 비교해 역전파 하는 방식으로 학습이 이루어짐

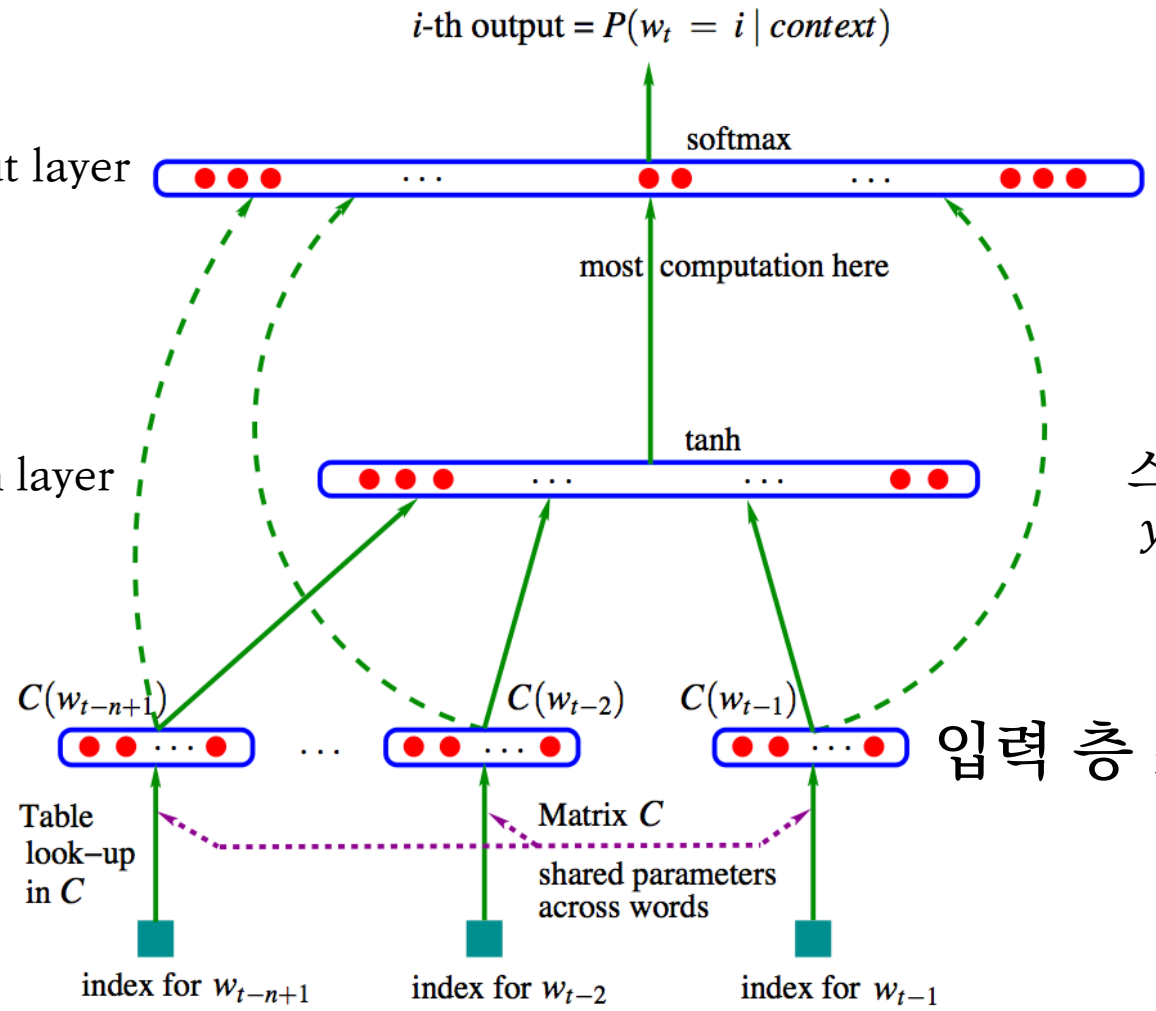
스코어 벡터 y 계산식

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

출력 층 output layer

은닉 층 hidden layer

입력 층 input layer



NPLM이 단어의 의미를 임베딩에 녹여내는 방법

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-gram의 n을 4 -> NPLM은 직전 3개 단어로 그 다음 단어 하나를 맞추는 과정에서 학습됨

Walking을 공유하는 3-gram

The cat is
A dog is
The dog was

The, A, cat, dog, is, was 등에 해당하는 C 행렬의 행 벡터들은 walking을 맞추는 과정에서 발생한 학습 손실 train loss를 최소화하는 그래디언트를 받아 동일하게 업데이트 됨

= 문맥 벡터가 벡터 공간에서 같은 방향으로 조금 움직인다는 이야기

NPLM이 단어의 의미를 임베딩에 녹여내는 방법

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-gram의 n을 4 -> NPLM은 직전 3개 단어로 그 다음 단어 하나를 맞추는 과정에서 학습됨

running을 공유하는 3-gram

A dog was
The cat is

The, A, cat, dog, is, was 가 벡터 공간에서 같은 방향으로 조금 움직임

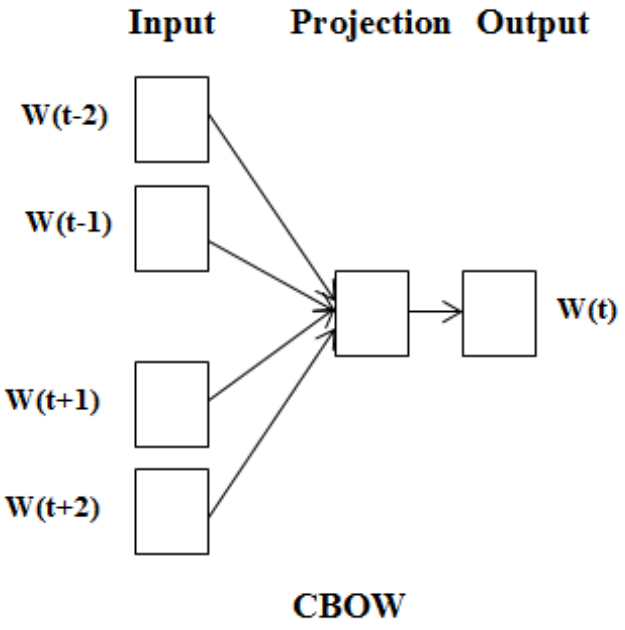
in 또한 마찬가지

이렇게 문장 내 모든 단어들을 한 단어 씩 훑으면서 말뭉치 전체를 학습하게 된다면 NPLM모델의 C행렬에 각 단어의 문맥 정보를 내재할 수 있게 됨

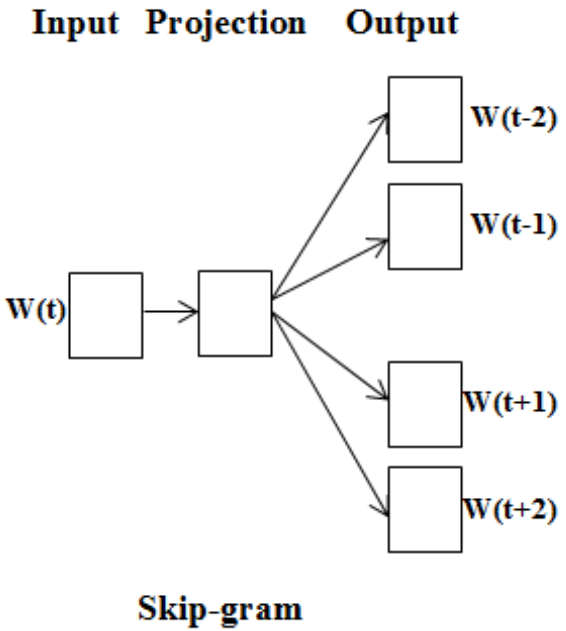
NPLM 정리

- NPLM은 그 자체로 언어 모델 역할 수행 가능
- 학습 데이터에 없는 문장의 등장 확률을 0으로 부여하는 n-gram 모델의 문제점
- 하지만 문맥이 비슷한 다른 문장을 참고해 확률 부여
- 결과적으로 NPLM은 새로운 문장을 다른 문장들과 비슷하게 추론하게 됨

Word2Vec 모델 기본 구조



CBOW



Skip-gram

{문맥 단어 4개, 타깃 단어}
입력, 출력 학습 데이터 1개 쌍

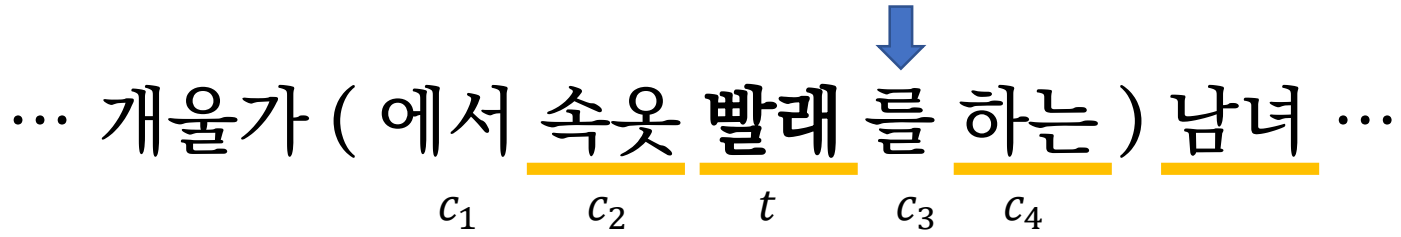
{타깃 단어, 타깃 직전 두 번째 단어}
{타깃 단어, 타깃 직전 단어}
{타깃 단어, 타깃 다음 단어}
{타깃 단어, 타깃 다음 두 번째 단어}
입력, 출력 학습 데이터 4개 쌍

같은 말뭉치로도 더 많은 학습 데이터 확보
-〉 임베딩 품질이 더 좋음

Word2Vec 학습하기 전 데이터 구축

- 포지티브 샘플
- 네거티브 샘플

- Skip-gram 모델: 전체 말뭉치를 단어별로 슬라이딩해 가며 학습 데이터 생성



- 소프트맥스 때문에 계산량이 비교적 큰 편
  정답 문맥 단어가 나타날 확률은 높이고 나머지 단어들 확률은 낮춰야 하는데 어휘 집합의 단어는 보통 수십만 개

Word2Vec 학습하기 전 데이터 구축

Skip- gram을 네거티브 샘플링

- 타깃 단어와 문맥 단어 쌍이 주어졌을 때 해당 쌍이 포지티브 샘플인지, 네거티브 샘플인지 이진 분류binary classification하는 과정에서 학습

✓1개의 포지티브 샘플과 k개의 네거티브 샘플만 계산
(= 매 스텝마다 시그모이드를 k+1회 계산)

k
- 작은 데이터 5~10
- 큰 데이터 2~5

VS

✓1스텝에 전체 단어를 모두 계산
(= 매 스텝마다 어휘 집합 크기만큼의 소프트맥스를 1회 계산)

Word2Vec 네거티브 샘플링

Vanilla Skip-Gram

W_output (old)				Learning R.		grad_W_output				W_output (new)		
-0.560	0.340	0.160	−	0.05	×	0.064	0.071	-0.014	=	**-0.563**	**0.336**	**0.161**
-0.910	-0.440	1.560				0.098	0.015	0.063		**-0.915**	**-0.441**	**1.557**
-1.210	-0.130	-1.320				0.069	0.089	0.045		**-1.213**	**-0.134**	**-1.322**
1.670	-0.150	-1.030				0.014	0.085	0.079		**1.669**	**-0.154**	**-1.034**
1.720	-1.460	0.730				-0.021	0.067	0.071		**1.721**	**-1.463**	**0.726**
0.000	1.390	-0.120				-0.098	-0.088	0.091		**0.005**	**1.394**	**-0.125**
-0.060	1.520	-0.790				-0.072	-0.078	-0.089		**-0.056**	**1.524**	**-0.786**
0.800	1.850	-1.670				0.046	-0.079	-0.053		**0.798**	**1.854**	**-1.667**
-1.370	1.320	-0.480				-0.049	-0.087	0.025		**-1.368**	**1.324**	**-0.481**
0.670	1.990	-1.850				-0.060	0.092	0.042		**0.673**	**1.985**	**-1.852**
-1.520	-1.740	-1.860				0.074	0.050	0.070		**-1.524**	**-1.743**	**-1.864**
(11X3)						(11X3)				(11X3)		

Negative Sampling

W_output (old)				Learning R.		grad_W_output				W_output (new)		
-0.560	0.340	0.160	−	0.05	×	Not computed!			=	-0.560	0.340	0.160
-0.910	-0.440	1.560								-0.910	-0.440	1.560
-1.210	-0.130	-1.320								-1.210	-0.130	-1.320
1.670	-0.150	-1.030								1.670	-0.150	-1.030
1.720	-1.460	0.730								1.720	-1.460	0.730
0.000	1.390	-0.120								0.000	1.390	-0.120
-0.060	1.520	-0.790								-0.060	1.520	-0.790
0.800	1.850	-1.670		Positive sample, w_o		0.031	0.030	0.041		**0.798**	**1.849**	**-1.672**
-1.370	1.320	-0.480		Negative sample, k=1		-0.090	0.031	-0.065		**-1.366**	**1.318**	**-0.477**
0.670	1.990	-1.850		Negative sample, k=2		0.056	0.098	-0.061		**0.667**	**1.985**	**-1.847**
-1.520	-1.740	-1.860		Negative sample, k=3		0.069	0.084	-0.044		**-1.523**	**-1.744**	**-1.858**
(11X3)						(11X3)				(11X3)		

Word2Vec 학습하기 전 데이터 구축

네거티브 샘플은 어떻게 뽑는 걸까

- Mikolov et al (2013b): 말뭉치에 자주 등장하지 않는 희귀한 단어가 네거티브 샘플로 조금 더 잘 뽑힐 수 있도록 설계

네거티브 샘플 확률

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

- $U(w_i)$: 해당 단어의 유니그램 확률(해당 단어 빈도/전체 단어 수)

Word2Vec 학습하기 전 데이터 구축

네거티브 샘플 확률

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

- $U(w_i)$: 해당 단어의 유니그램 확률(해당 단어 빈도/전체 단어 수)

예) 말뭉치 단어; 용, 미르 / 구성비율 각 0.99, 0.01

- 네거티브로 뽑힐 확률

$$P(\text{용}) = \frac{0.99^{0.75}}{0.99^{0.75} + 0.01^{0.75}} = 0.97$$

$$P(\text{미르}) = \frac{0.01^{0.75}}{0.99^{0.75} + 0.01^{0.75}} = 0.03$$

=〉 등장할 빈도가 더 희귀한 미르의 뽑힐 확률이 살짝 높아짐

Word2Vec 학습하기 전 데이터 구축

서브샘플링

자주 등장하는 단어는 학습에서 제외하여 학습량을 효과적으로 줄여 계산량을 감소시키는 전략

서브샘플링 확률

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

- f(w_i): w_i의 빈도, t: 하이퍼 파라미터(10^-5)

* 은/는: f(w_i)가 0.01로 빈도가 높은 단어는 확률(P)이 0.9684나 되어 100번의 학습 기회 중 96번 정도는 학습에서 제외, 반대로 확률이 0에 가까울 땐 해당 단어가 나올 때마다 빼놓지 않고 학습 시키는 구조

Word2Vec 모델 학습

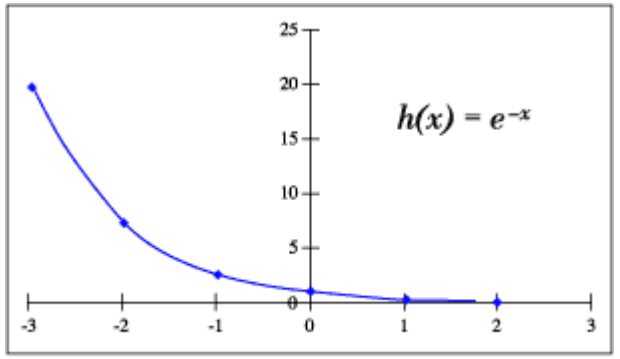


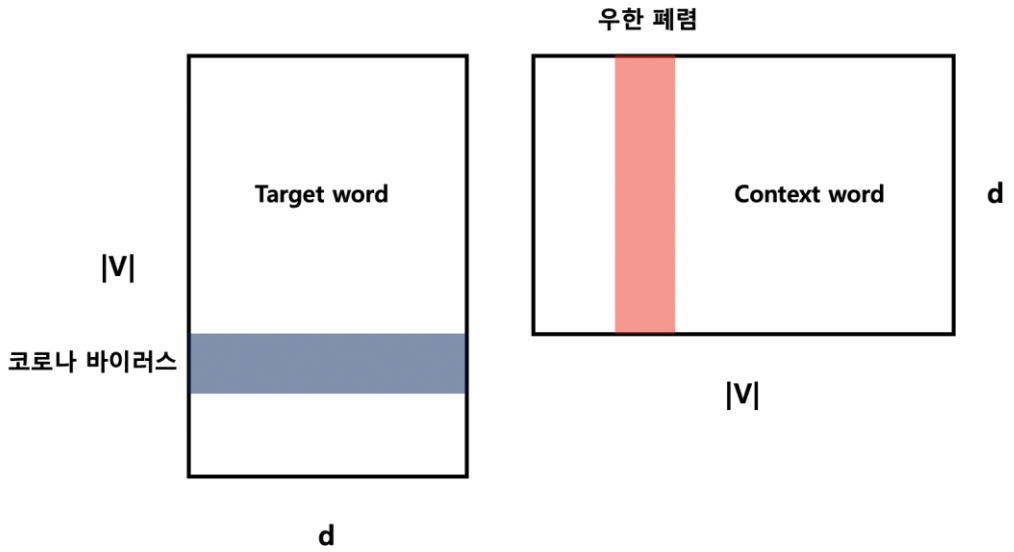
Figure 3

Skip-gram 모델: 이진 분류를 예측하는 과정에서 학습

타깃 단어와 문맥 단어 쌍이 실제 포지티브 샘플이라면

Maximize $P(+|t,c) = \frac{1}{1+\exp(-u_t v_c)}$ => 내적 값 ⬆

모델의 학습 파라미터는 U와 V행렬 두 개 뿐(크기: |V|*임베딩 차원수(d))



Word2Vec 모델 학습

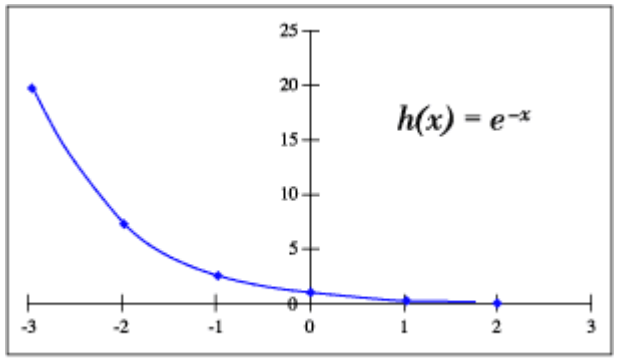


Figure 3

Skip-gram 모델: 이진 분류를 예측하는 과정에서 학습

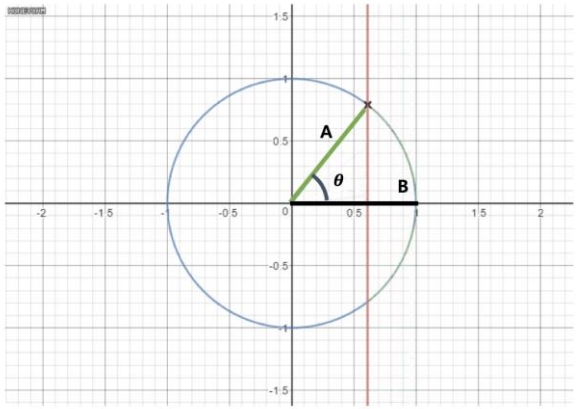
타깃 단어와 문맥 단어 쌍이 실제 네거티브 샘플이라면

Maximize $P(-|t,c) = 1 - P(+|t,c) = \frac{\exp(-u_t v_c)}{1+\exp(-u_t v_c)}$ => 내적 값 ⬇

모델의 학습 파라미터는 U와 V행렬 두 개 뿐(크기: |V|*임베딩 차원수(d))

두 벡터의 내적은 코사인 유사도와 비례

- 내적 값 상향: t와 c에 해당하는 단어 벡터 간 유사도를 높인다
- 내적 값 하향: t와 c에 해당하는 단어 벡터 간 유사도를 낮춘다



Word2Vec 모델 학습

Skip-gram 모델: 이진 분류를 예측하는 과정에서 학습

모델의 손실함수를 알았으니 최대화하는 파라미터 MLE구함

로그우도 함수 log-likelihood function

$$\text{Maximize } \mathrm{L}(\theta) = \log P\left(+|t_p, c_p\right) + \sum_{i=1}^{k} logP(-|t_{n_i}, c_{n_i})$$

- 모델 파라미터인 θ를 한 번 업데이트할 때 1개 쌍의 포지티브 샘플과 k개 쌍의 네거티브 샘플이 학습된다는 의미
- 최대화하는 과정에서 skip-gram 모델은 말뭉치의 분포 정보를 단어 임베딩에 함축시키게 됨
- 분포가 유사한 단어 쌍은 그 속성 또한 공유할 가능성이 높음(유의 관계, 반의 관계, 상하 관계)
- 모델 학습이 완료되면 U($target_{word}$에 관한 행렬)만 d차원의 단어 임베딩으로 사용할 수도 있고, $U + V^t$ 행렬 혹은 concatenate([U, V^t])를 사용할 수도 있다.

Word2Vec 튜토리얼

```
>>> from gensim.test.utils import common_texts, get_tmpfile
>>> from gensim.models import Word2Vec
>>>
>>> path = get_tmpfile("word2vec.model")
>>>
>>> model = Word2Vec(common_texts, size=100, window=5, min_count=1, workers=4)
>>> model.save("word2vec.model")
```

- size: Word2Vec 임베딩 차원수
- workers: CPU 스레드 개수
- sg: skip-gram 모델인지 여부를 나타내는 하이퍼 파라미터
  1이면 skip-gram, 0이면 CBOW 모델로 학습

https://radimrehurek.com/gensim/models/word2vec.html

FastText 모델 기본 구조

각 단어를 문자 단위 n-gram으로 표현, 이 외에는 Word2Vec과 동일

‘시나브로’의 n=3인 문자 단위 n-gram

〈시나, 시나브, 나브로, 브로〉, 〈시나브로〉

- 〈〉: 단어의 경계를 나타내 주는 특수 기호

단어의 임베딩을 단어 벡터의 합으로 표현

$$u_{\text{시나브로}} = z_{\langle\text{시나}} + z_{\text{시나브}} + z_{\text{나브로}} + z_{\text{브로}\rangle} + z_{\langle\text{시나브로}\rangle}$$

$$u_t = \sum_{g \in G_t} z_g$$

G_t={〈시나, 시나브, 나브로, 브로〉, 〈시나브로〉}=타깃 단어 t에 속한 문자 단위 n-gram 집합

FastText 모델 학습

네거티브 샘플링

$$\text{Maximize } P(+|t,c) = \frac{1}{1+\exp(-u_t v_c)} = \frac{1}{1+\exp\left(-\sum_{g\in G_t} {z_g}^T v_c\right)}$$

-〉 확률(t,c가 포지티브 샘플일 확률)을 최대화하는 과정에서 학습

이때, 타깃 단어(t), 문맥 단어(c) 쌍을 학습할 때 타깃 단어(t)에 속한 문자 단위 n-gram 벡터(z)들을 모두 업데이트 함

즉, 만약

시나브로(타깃 단어(t)), 쌓였다(문맥 단어의 포지티브 샘플(c)) 이라면

〈시나, 시나브, 나브로, 브로〉, 〈시나브로〉 등 문자 n-gram 벡터(z)들 각각을 쌓였다에 해당하는 단어 벡터(v_c)와의 유사도를 높인다

네거티브 샘플로 '컴퓨터'일 경우엔?

FastText 모델의 강점

조사나 어미가 발달한 한국어에 종은 성능을 보임

용언(동사, 형용사)의 활용이나 그와 관계된 어미들이 벡터 공간상 가깝게 임베딩 됨

하였다(타깃 단어(t)), 진행(문맥 단어의 포지티브 샘플(c))

〈하였, 하였다, 였다〉 벡터(z)들 각각이 진행에 해당하는 벡터(v_c)와의 유사도가 높아짐

하였다 벡터와 하(다), 했(다), (하)였으며 등에 해당하는 벡터간 유사도가 높을 것
-〉 문맥이 서로 비슷하므로

FastText 모델의 강점

오타나 미등록 단어에도 로버스트함

-> 각 단어의 임베딩을 문자 단위 n-gram 벡터의 합으로 표현하므로

서울특별시

서울 등이 문자 단위 n-gram에 포함

울특, 특별 등이 모두 미등록 단어라 할지라도 서울특별시에 대한 임베딩 추정 가능

FastText 모델의 강점

한글을 자소 단위로 분해해 자소 각각을 하나의 문자로 보고 문자 단위 n-gram 임베딩 수행

코사인 유사도가 가장 높은 단어들을 뽑는 코드 프로세스

-> 임베딩을 만들 때 자소 단위로 분해한 말뭉치 사용

-> 어휘 집합에 속한 단어들 또한 자소로 분리

-> most_similar함수에 인자로 넣을 쿼리 단어 또한 자소 단위로 분해한 뒤 FastText 임베딩 추출

-> 전체 단어 목록의 코사인 유사도를 구해 상위 단어 추림

-> 이 단어들의 자소를 합쳐 다시 원래 단어로 복원 시킴

https://radimrehurek.com/gensim/models/fasttext.html