Word2vec

1. Introduction

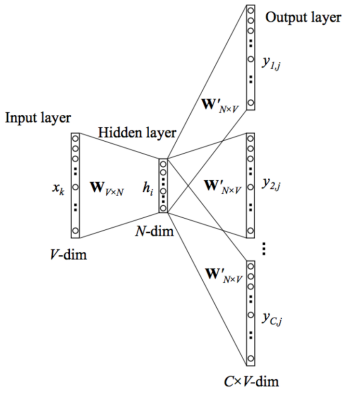
word2vec은 skip-gram model을 확장한 기법이다. skip-gram model은 매우 많은 단어를 효율적인 계산을 통해 벡터로 나타내는 기법으로, NNLM(Neural Network Language Model)의 단점인 non-linear hidden layer으로 인한 complexitiy을 보완한다. word2vec은 여기에 sub-sampling을 추가하여 더 빠른 속도를 낼 수 있음을 보인다.

단어 표현은, 단어와 단어의 단순한 합이 또 다른 단어를 의미하지 않을 수도 있다. 예를들어 밤낮이라는 단어를 생각해보자. 밤낮은 말 그대로 밤과 낮을 의미할 수도 있지만 이보다는 '밤낮으로 바빴다.'라는 의미로 더 많이 쓰인다. 논문에서는 단어 단위 벡터의 이러한 한계를 지적하며 phrase 단위 벡터로 나아갈 것을 제안한다. 단어 기반에서 구문 기반으로의 전환은, 간단하게 데이터 기반 접근법으로 아주 큰 데이터 세트로부터 구문을 파악하고 이를 마치 하나의 token으로 취급하는 방법이 있다. 또한 skip-gram model의 흥미로운 점으로부터 이러한 확장을 할 수도 있는데, korea을 나타내는 벡터와 capital을 나타내는 벡터를 합쳐서(concatenate) seoul을 나타내는 벡터를 만들어낼 수도 있다.

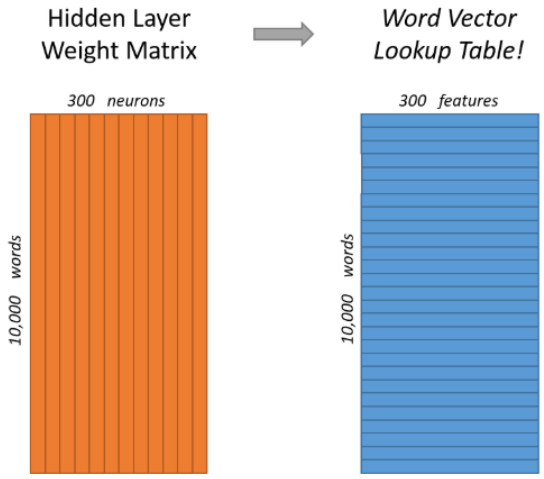
2-(1) word2vec의 아키텍처 : Skip gram model

word2vec의 학습 parameter

word2vec에 대해서 자세히 알아보기 전에 word2vec이 Skip-gram model에 기반하여 만들어졌기 때문에 이에 대해서 자세히 알아보자.



word2vec의 아키텍처는 은닉층이 하나인 간단한 뉴럴네트워크 구조이다. W와 $W^{'}$의 두 가중치 행렬이 위 구조의 핵심인데, word2vec의 학습결과가 이 두 행렬이기 때문이다. 입력층과 은닉층을 잇는 가중치 행렬 W에 대해 자세히 살펴보자. W는 $V \times N$ 행렬로, V는 임베딩하려는 단어의 수, N은 은닉층의 노드 개수, 즉 사용자가 지정하는 임베딩 벡터의 차원이다. word2vec은 최초 입력으로 $1 \times V$의 one-hot-vector을 받으므로 입력층과 은닉층을 잇는 행렬은 $V \times N$ 차원이 되어야 한다. 예를 들어, 학습 말뭉치 단어가 1만개 있고 은닉층 노드를 300으로 지정했다고 하자. 즉, $V = 10000, N = 300$이므로 W은 10000×300 차원의 행렬이 될 것이다. 이는 아래 그림의 왼쪽 행렬 형태이다.



여기서 흥미로운 점은, W가 입력벡터인 one-hot-encoding 벡터와 은닉층을 이어주는 가중치 행렬임과 동시에 word2vec의 최종 결과물인 임베딩 단어 벡터의 모음이기도 하다는 것이다.
단어가 5개 뿐인 말뭉치에서 네 번째 단어와 가중치 행렬 W을 곱한다고 생각해보자.

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

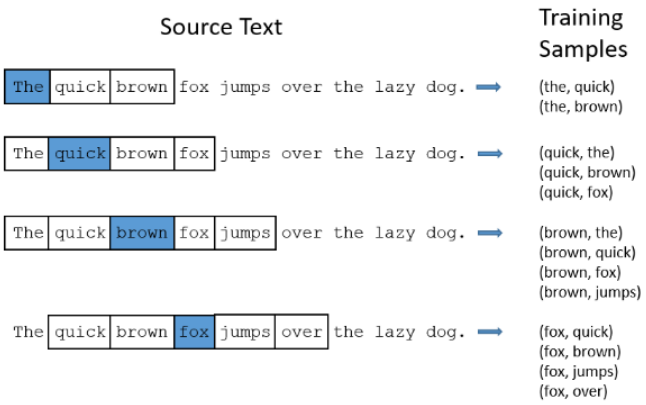
입력벡터가 one-hot-encoding 벡터이기 때문에 가중치 행렬의 네 번째 행만을 참조(lookup)한다. 따라서 만약에 W의 학습이 종료되고 최종적으로 output 벡터를 만들기 위해 XW을 한다면, 여기서 X는 one-hot-encoding으로 구성된 행렬이므로 $XW = W$가 될 것이다.

word2vec의 계산량

- 현재 단어를 hidden layer에 보내는데에 N
- Ouptut을 계산하는데에 $N \times V$, 계산량을 줄이는 hierarchical softmax 등을 이용한다면 $N \times log\, V$
- 총 C개의 단어에 대해 진행해야 하므로 C배
- 위의 논의를 종합하면 $C(N + N \times log\, V)$ 또는 $C(N + N \times V)$

word2vec의 벡터 생성 과정

'The quick brown fox jumps over the lazy dog.' 문장으로 시작하는 학습말뭉치가 있다고 하자. 윈도우(한번에 학습할 단어 개수) 크기가 2인 경우 word2vec의 아키텍처가 받는 입력과 정답은 아래와 같다.



학습의 첫 단계를 살펴보자. 중심 단어가 The이고 윈도우의 크기가 2인데, The 앞쪽에 단어가 없으므로 뒤쪽 단어 두 개인 quick, brown을 학습에 사용한다. 여기서 주목해야할 점은, training samples을 만들 때, (The, quick, brown)으로 만드는 것이 아니라 quick과 brown을 따로 떼어서 (The, quick), (The, Brown)으로 만든다는 것이다.

이렇게 첫 단계가 끝나면 The의 옆 단어인 quick으로 옮긴다. 이 때는 크기가 2인 윈도우를 기준으로 주변 단어가 The, brown, fox이므로 이를 이용하여 총 3개의 training samples을 만든다.

이런 식으로 말뭉치 내에 존재하는 모든 단어를 윈도우 크기로 슬라이딩 해가며 학습을 하면 iteration 1회가 마무리 된다.

이처럼 skip gram은 중심 단어의 벡터를 업데이트할 기회가 크기가 2인 윈도우 기준으로 4번이나 있다. 반면에, CBOW는 크기가 2인 윈도우 기준으로 주변 네 개 단어로 중심 단어를 한 번만 업데이트 한다. 따라서 CBOW에 비해 skip gram의 성능이 일반적으로 더 좋음이 알려져 있다.

word2vec의 학습

Skip-gram의 objective function은 아래와 같다. c는 중심단어, o는 주변단어를 의미한다. 즉, 중심 단어가 주어졌을 때, 주변 단어가 나타날 확률을 softmax로 구하고, 이를 최대화하는 방향으로 학습을 진행한다.

$$P(o \mid c) = \frac{exp\left(u_o^T v_c\right)}{\sum_{w=1}^{W} exp\left(u_w^T v_c\right)} \tag{1}$$

우변의 v는 입력층-은닉층을 잇는 가중치 행렬 W의 행벡터, u는 은닉층-출력층을 잇는 가중치 행렬 W'의 열벡터이다.

(1)을 최대화하기 위해서 분자는 최대로, 분모는 최소로 만드는 것이 좋을 것이다. 분자는 중심단어와 주변단어의 내적 값인데, 벡터 내적은 코사인이므로 내적 값 향상은 단어 벡터 간 유사도를 높인다는 의미로 이해하자.

분모는 윈도우 크기 내에 등장하지 않는 단어들과 중심 단어의 내적 값인데, 이를 최소화한다는 것은 유사도를 감소한다는 의미로 이해하자.

최적(optimized)의 중심 단어를 구하기 위해 (1)에 대한 v_c의 gradient을 구하고 gradient descent

방식을 이용하여 update하자. v_c에 대한 gradient는 아래와 같다.

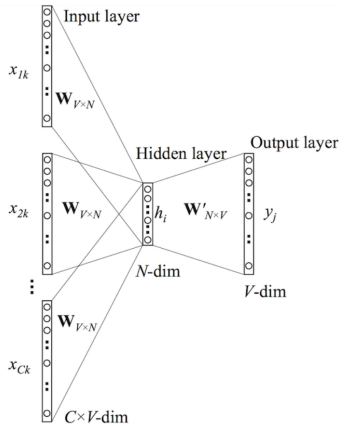
$$\begin{aligned}
\frac{\partial}{\partial v_c} log\, P(o \mid c) &= \frac{\partial}{\partial v_c} log\, \frac{exp\left(u_o^T v_c\right)}{\sum_{w=1}^{W} exp\left(u_w^T v_c\right)} \\
&= \frac{\partial}{\partial v_c} u_o^T v_c - \frac{\partial}{\partial v_c} log \sum_{w}^{W} exp(u_w^T v_c) \\
&= u_o^T - \frac{1}{\sum_{w}^{W} exp(u_w^T v_c)} \left(\sum_{w}^{W} exp(u_w^T v_c) \cdot u_w \right) \\
&= u_o^T - \sum_{w=1}^{W} \frac{exp\left(u_w^T v_c\right)}{\sum_{q=1}^{W} exp\left(u_q^T v_c\right)} \cdot u_w \\
&= u_o^T - \sum_{w=1}^{W} P(w \mid c) \cdot u_w
\end{aligned} \tag{2}$$

(2)에서 구한 v_c에 대한 gradient을 통해 v_c를 아래와 같이 업데이트한다.

$$v_c^{t+1} = v_c^t + \alpha \left(u_o^T - \sum_{w=1}^{W} P(w \mid c) \cdot u_w \right) \tag{3}$$

2-(2) CBOW

Skip gram과 반대의 개념을 CBOW을 간단히 살펴보자. CBOW의 모델의 아키텍쳐는 아래와 같다.



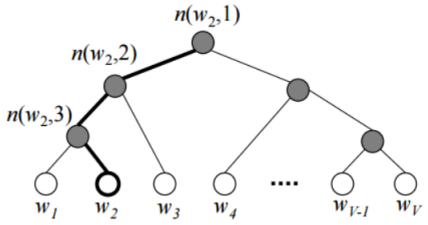
Skip-gram 과는 반대로, '코딩과 통계는 _를 잘하기 위한 필수 역량입니다.' 라는 문장에서 주변의 말을 통해 _에 들어갈 말을 유추한다. CBOW 모델에서 하나의 단어를 처리할 때 드는 계산량은 아래와 같다.

- C개의 단어를 hidden layer에 보내는데에 $C \times N$
- hidden layer에서 output layer로 가는 데에 $N \times V$, 계산량을 줄이는 hierarchical softmax 등을 이용한다면 $N \times log\, V$
- 위의 논의를 종합하면 총 $C \times N + N \times V$, 또는 $C \times N + N \times log\, V$

앞서 말한 Skip-gram이나 CBOW는 V의 크기가 일반적으로 매우 크므로 생각보다 학습이 오래 걸린다. Ouput layer에서 softmax을 통해, 확률을 계산하는데 각 단어에 대해 전부 계산을 해서 normalization을 해주어야 하고 이에 따라 추가적인 연산이 발생하기 때문이다. 이러한 문제점에 대한 대안으로 계산량을 줄이는 방법들이 개발되었고 Hierarchical softmax와 negative sampling이 그것인데, Skip-gram 모델에 이 두 방법을 적용한 모델이 바로 word2vec 모델인 것이다.

2-(3) Hierarchical Softmax

이 방법은 각 단어들을 leaves로 가지는 binary tree을 가정한다. 그리고 각 leaf(단어)의 확률을 계산할 때 root에서부터 해당 leaf로 가는 path에 따라서 확률을 곱해나가는 식으로 해당 단어가 나올 최종적인 확률을 계산한다.



Hierarchical softmax는 주변 단어가 주어졌을 때, 단어의 확률을 아래와 같이 정의한다.

$$p(w \mid w_I) = \prod_{j=1}^{L(w)-1} \sigma\left(I\left[n(w,j+1) = ch(n(w,j))\right] \cdot v'^{T}_{n(w,j)} v_{w_I}\right) \tag{4}$$

notation을 아래와 같이 정리하였다.

- $n(w, j)$: jth node on the path from the root to w
- $L(w)$: length of this path, so $n(w,1) = root,\ n(w, L(w)) = w$
- $ch(n)$: arbitrary fixed child of n
- $I\left[\cdot\right]$: be 1 if $\cdot$ is true, -1 otherwise.
- $\sigma(x)$: sigmoid function.

(4)와 같이 단어의 확률을 정의한다면, $p(w \mid w_I)$을 계산하기 위해서, 각 스텝마다 길이 N의 벡터 두 개의 내적을 해야하므로 계산량 N, 평균적으로 루트로부터 leaf까지의 거리는 $log\,V$일 것이므로 $N \times log\,V$만으로 특정 단어가 나올 확률을 계산할 수 있다고 한다.
또한 각 노드마다 왼쪽 child으로 갈 확률이 $\sigma(x)$, 오른쪽 자식으로 갈 확률이 $\sigma(-x) = 1 - \sigma(x)$이기 때문에, (1)과 같이 softmax function에서 normalization을 해줄 필요가 없어서 V만큼의 계산이 덜 발생함을 확인할 수 있다.

3. Reducing Computation Further

(1) Subsampling frequent words

word2vec은 결국 (1)을 최대화하는 방향으로 W와 $W^{'}$가 만들어지는데, 이는 단어 수가 늘어날수록 계산량이 매우 커지는 구조이다. 따라서 word2vec은 모든 단어들이 나타날 때마다 학습을 하는 것이 아니라, 자주 등장하는 단어, 예를 들어 the, a, in의 학습량을 확률적인 방식으로 줄이는 방식을 제안하였다. 논문에서는 i번째 단어인 w_i를 학습에서 제외시키기 위한 확률을 아래와 같이 정의하였다.

$$P(w_i) = 1 - \sqrt{\frac{t}{f(w_i)}} \tag{3}$$

(3)에서 $f(w_i)$는 w_i가 말뭉치에서 등장한 비율 (count / total count)를 말하고 t는 사용자가 지정해주는 값인데, 0.00001을 권장하고 있다. 예를 들어, '은/는'과 같은 조사들의 등장 비율을 0.05라고 한다면 (3)을 이용한 $P(w_i)$은 0.985나 되기 때문에 100번의 학습 기회 가운데 99번 정도는 학습에서 제외하게 된다. 반대로 등장 비율이 적어 $P(w_i)$가 작다면, 그 단어를 제외시키는 횟수가 작을 것이다. 이와 같이 subsampling은 단어마다 등장 비율에 기반해, 학습량을 줄여 계산량을 감소시키는 방법이다.

(2) Negative Sampling

Negative Samling은 hierarchical softmax의 한 대안이다(파이썬에서 word2vec을 할 때, 이 둘 중 하나만 선택해야 한다.) softmax의 계산량이 너무 많아서 hierarchical softmax을 사용했음을 생각하면, Negative Sampling의 목적도 자연스럽게 유추할 수 있다. 즉, softmax에서 확률을 구하려면 특히 분모의 계산량이 너무 큰데, 즉 중심단어와 나머지 모든 단어에 대해 내적을 한 뒤에 다시 exp을 취해야 한다. 보통, 전체 단어가 10만개이기 때문에 계산량이 매우 커진다. 이 때문에, 전체 단어가 아니라 일부 단어만 뽑아서 계산을 하는 것이 바로 Negative Sampling의 아이디어이다.
Negative Sampling의 절차는 다음과 같다.

- 사용자가 지정한 윈도우 사이즈 내에 등장하지 않은 단어(negative sample이라고 부른다)를 $P(w_i) = \frac{f(w_i)^{3/4}}{\sum_{j=1}^{n} f(w_j)^{3/4}}$의 확률로 5~20개 정도 뽑는다.
- 이를 정답단어와 합쳐 전체 단어처럼 softmax 확률을 구한다.
- 즉, 윈도우 사이즈가 10일 경우, 최소 15개에서 최대 30개 단어를 대상으로만 softmax 확률을 계산하고 파라미터 업데이트도 이 30개 대상으로만 이루어진다.

논문에서는 negative sample의 개수 k를 작은 데이터에 대해서는 5개에서 20개, 큰 데이터에 대해서는 2개에서 5개로 할 것을 권장하고 있다.

Reference

1. http://dsp.yonsei.ac.kr/139926
2. https://ratsgo.github.io/from%20frequency%20to%20semantics/2017/03/30/word2vec/
3. Distributed Representations ofWords and Phrases and their Compositionality (Thomas Mikolov)