**텍스트 전처리**

자연어 처리에서 크롤링 등으로 얻어낸 corpus data가 필요에 맞게 전처리되지 않은 상태라면, 해당 데이터를 사용하고자 하는 용도에 맞게 토큰화, 정제, 정규화 작업을 해야한다.

1. 토큰화

보통 단어를 기준으로 하는 토큰화를 많이 한다. 이번 캡스톤디자인에서는 한글 데이터를 기반으로 할 예정이기 때문에 한글 문장의 형태소를 분리해주는 KoNLPy 라이브러리를 사용해 단어를 분리해 사용하는 것이 좋을 것 같다.

KoNLPy에는 Okt, Mecab, Komoran, Hannanum, Kkma라는 형태소 분석기가 있다.

결국 이는, 단어 토큰화가 아니라 형태소 토큰화라고 볼 수 있다. okt와 kkma를 이용한 토큰화가 널리 퍼져있는 것으로 보인다.

2. 정제, 정규화

정제: 갖고 있는 corpus data로부터 노이즈 데이터를 제거

정규화: 표현 방법이 다른 단어들을 통합시켜 같은 단어로 만들어 줌.

**언어 모델(Language Model, LM)**

언어라는 현상을 모델링하고자 단어 시퀀스에 확률을 할당하는 모델.

크게 통계를 이용한 방법과 인공 신경망을 이용한 방법으로 구분 가능(최근은 인공 신경망이 성능이 더 좋음).

인공 신경망 -> GPT, BERT 등

단어 시퀀스에 확률을 할당하기 위해 가장 보편적으로 사용되는 방법은 언어 모델이 이전 단어들이 주어졌을 때 다음 단어를 예측하도록 하는 것.

**통계를 이용한 언어 모델(Statistical Language Model, SLM)**

각 단어는 문맥이라는 관계로 인해 이전 단어의 영향을 받아 나온 단어 <= 핵심 포인트

그렇기 때문에, 문장의 확률을 구하고자 조건부 확률을 이용

밑의 예시는 문장 ‘An adorable little boy is spreading smiles’라는 문장에 대해 조건부 확률을 이용해 문장의 확률을 구한 것 => 각 단어들이 이전 단어가 주어졌을 때 다음 단어로 등장할 확률의 곱으로 구성됨.

=> 문장의 확률을 구하기 위해 다음 단어에 대한 예측 확률을 모두 곱한다!

그렇다면, SLM은 이전 단어로부터 다음 단어에 대한 확률은 어떻게 구하나? -> 카운트에 기반해 확률 계산

=> A가 100번 등장, A 다음에 B가 30번등장 => P(B|A) =30% 이런 식의 카운트 기반 계산

그런데, 이러한 방식은 정말 방대한 양의 데이터를 필요로 함. -> 희소 문제 발생.

이를 완화하기 위한 해결책? -> N-GRAM, SMOOTHING, BACK-OFF 등의 여러가지 일반화 기법… 그러나 근본적인 해결책이 되지는 못함. 이 때문에 트렌드가 인공 신경망 언어 모델로 넘어감.

**N-gram 언어 모델**

모든 단어를 고려하는 것이 아닌, 일부 단어만 고려하는 접근 방법을 사용하는 SLM의 일종.

SLM의 한계? -> 현재 가지고 있는 훈련 corpus data에 확률을 계산하고 싶은 문장이나 단어가 없을 수 있다. 그런데, 참고하는 앞선 단어들을 줄인다면? => 카운트할 가능성 증가.

ex) P(is|An adorable little boy)를 P(is|boy)와 유사하게!

위의 예시는 n-gram의 n이 1인 것. 즉, N-gram은 임의의 단어 개수를 정하기 위한 기준!

n-gram: n개의 연속적인 단어 나열

n-gram을 통한 언어 모델에서는 다음에 나올 단어의 예측은 오직 n-1개의 단어에만 의존.

but, 이것도 한계점이 있다. 앞의 몇 단어만 보니까 의도하고 싶은 대로 문장을 끝맺음 짓지 못하는 경우가 생길 수 있다.

ex) 아주 멋진 소년이 ~~~~~~~~~~~~~~~~(욕을 했다/웃음 지었다). <- 예측이 힘들다.

=> n-gram의 n을 선택하는 것은 trade-off문제.

**한국어에 적용하는 언어 모델**

한국어의 특징들은 여러가지가 있다.

1. 어순이 크게 중요하지 않다.

단어 순서를 뒤죽박죽으로 바꾸어도 의미가 전달이 되기 때문에, 확률에 기반한 언어 모델이 제대로 다음 단어를 예측하기 어렵다.

2. 한국어는 교착어이다.

조사가 많아 어절 단위로 토큰화를 하면 경우의 수가 너무 많다! ex) 그가, 그를, 그의, 그에게 등

3. 띄어쓰기가 제대로 지켜지지 않음.

띄어쓰기를 안 해도 의미가 잘 전달되고, 규칙 또한 까다로운 언어다.

이 세 가지 특징들을 잘 인지하고 언어 모델을 작성해야 한다.

**단어의 표현 방법**

1. 국소 표현

각 단어에 1, 2, 3과 같이 숫자를 매핑해 부여하는 것. 이산 표현이라고도 한다.

2. 분산 표현

단어 근처에 자주 등장하는 단어를 가지고 해당 단어의 느낌을 설명 -> 단어의 뉘앙스를 표현할 수 있게 된다. 연속 표현이라고도 함.

**Bag of Words (BOW)**

단어들의 순서는 전혀 고려하지 않고, 단어들의 출현 빈도에만 집중하는 텍스트 데이터의 수치화 표현 방법.

Bag of Words? = 단어들의 가방

scikit-learn 라이브러리가 영어에 대해서 손쉽게 BoW를 만들 수 있는 메서드인 CountVectorizer를 지원한다.

from sklearn.,feature\_extraction.text import CountVectorizer

=> 띄어쓰기만을 기준으로 단어를 자르는 낮은 수준의 토큰화 진행. 따라서 한국어에 적용하려면 형태소 분석이 필요할 듯?

불용어(stop\_words)

**문서 단어 행렬(DTM)**

서로 다른 문서들의 BoW들을 결합한 표현 방법인 문서 단어 행렬.

문서들을 서로 비교할 수 있도록 수치화할 수 있다는 점에서 의의를 가짐.

**TF-IDF 기법**

TF(Term Frequency, 단어 빈도): 특정 문서에서 나타나는 특정 단어의 총 횟수

DF(Document Frequency, 문서 빈도): 특정 단어가 나타나는 문서의 수

IDF(Inverse Document Frequency, 역문서 빈도): DF에 반비례하는 수, 그러나 단순한 역수가 아니라, 로그를 취해주고, 분모에 1을 더해준다.

왜 로그? -> 총 문서의 수 N이 커질수록 IDF의 값이 기하급수적으로 커질 것이기 때문.

왜 1을 더해? -> 특정 단어가 전체 문서에서 등장하지 않을 경우 분모가 0이 되는 상황을 방지하기 위함.

TF-IDF는 모든 문서에서 자주 등장하는 단어는 중요도가 낮다고 판단, 특정 문서에서만 자주 등장하는 단어는 중요도가 높다고 판단.

=> TF-IDF값이 낮으면 중요도가 낮음. 높으면 중요도가 높음.

따라서, a나 the 같은 불용어는 TF-IDF값이 낮다.

scikit-learn 라이브러리에서 쉽게 사용할 수 있다.

from sklearn.feature\_extraction.text import TfidfVectorizer

**문서의 유사도**

문서의 유사도의 성능은 각 문서의 단어들을 어떤 방법으로 수치화해서 표현하는지, 문서간의 단어들의 차이를 어떤 방법으로 계산헀는지에 달려있다.

1. 코사인 유사도

두 벡터 간의 코사인 각도를 이용해 구할 수 있는 두 벡터의 유사도를 의미함.

완전 동일 = 1, 90도 = 0, 180도 반대 = -1 => -1 ~ 1의 값을 가짐 => 1에 가까울수록 유사도가 높다!

numpy를 통한 코사인 유사도 구하기

dot(A, B) / (norm(A) \* norm(B))

코사인 유사도는 문서의 길이가 다른 상황에서 비교적 공정한 비교를 할 수 있도록 도와줌.

왜? => 벡터의 크기가 아닌, 벡터의 방향(패턴)에 초점을 두기 때문이다.

2. 유클리드 거리

문서의 유사도에서는 잘 사용되지 않음…

3. 자카드 유사도

**피드 포워드 신경망 언어 모델(NNLM)**

앞서 말했던 희소문제(데이터 불충분)은 기계가 단어 간 유사도를 알 수 있다면 해결할 수 있는 문제임.

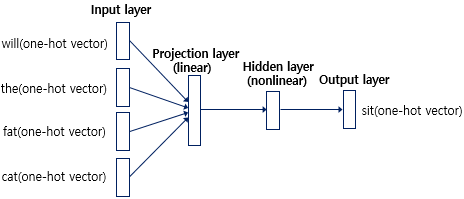
따라서, 언어 모델이 단어의 유사도 또한 학습할 수 있도록 설계한다면, 훈련 데이터에 없는 단어 시퀀스에 대한 예측이라도 유사한 단어가 사용된 단어 시퀀스를 참고해 보다 정확한 예측 가능.

=> 해당 아이디어를 가지고 설계한 언어 모델이 NNLM, 이것은 word embedding의 아이디어이기도 함.

가장 먼저 해야할 일 => 기계가 단어를 인식할 수 있도록 모든 단어를 숫자로 인코딩하는 것(one-hot encoding)

what = [1, 0, 0, 0, 0, 0, 0] will = [0, 1, 0, 0, 0, 0, 0] the = [0, 0, 1, 0, 0, 0, 0] fat = [0, 0, 0, 1, 0, 0, 0] cat = [0, 0, 0, 0, 1, 0, 0] sit = [0, 0, 0, 0, 0, 1, 0] on = [0, 0, 0, 0, 0, 0, 1] => 크기가 7인 차원을 가지는 one-hot vector들

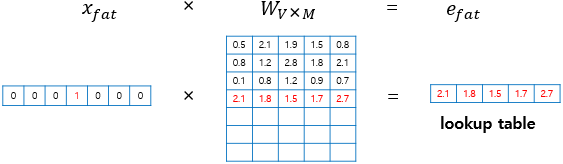
이 벡터들이 훈련을 위한 NNLM의 입력이면서, 예측을 위한 레이블이 된다. 즉, ‘what will the fat cat’을 입력받아 ‘sit’을 예측하는 일은 what, will, the, fat, cat의 vector를 입력받아 sit의 벡터를 예측하는 문제가 되는 것이다.

(NNLM의 구조)

NNLM은 N-GRAM 언어 모델과 유사하게 다음 단어 예측 시, 정해진 N개의 단어만을 참고함. 위의 예시는 N=4인 경우이다. 해당 범위를 ‘윈도우’라고도 함.

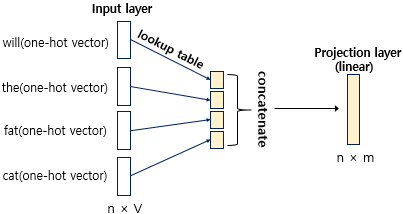
4개의 벡터를 입력 받은 NNLM은 다음 층인 투사층(Projection layer)을 통과함. 이 레이어는 신경망에서의 은닉층과 다르게 활성함수(activation function)이 존재하지 않고, 가중치 행렬과의 연산만 이루어짐.

투사층의 크기가 M이라면, 각 입력 단어들은 투사층이세 V x M 크기의 가중치 행렬과 곱해짐(V=단어 집합의 크기, 여기서는 7)



입력 벡터가 원-핫 벡터이기 때문에 결국 가중치 행렬의 원-핫 벡터가 1인 인덱스의 값을 그대로 가져오는 행위와 동일. 때문에 이 작업을 lookup table이라고도 함.

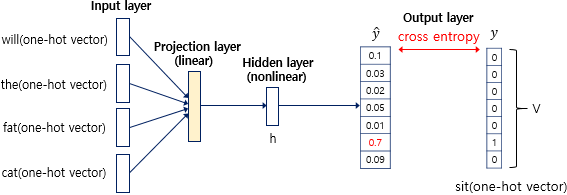
위 작업을 거치면, 원-핫 벡터는 V보다 차원이 작은 M차원의 단어 벡터로 mapping됨. 이 mapping된 벡터를 embedding vector라고 함.



각 단어를 임베딩 벡터로 변환하고, 이 값들은 모두 concatenate된다.

x를 각 원-핫 벡터, NNLM이 예측하고자 하는 단어가 문장에서 t번째 단어, 윈도우의 크기를 n이라 했을 때, 투사층을 식으로 표현하면 아래와 같다(세미 콜론은 concatenate)

이후, 투사층의 결과는 h의 크기를 가지는 은닉층을 지난다. 은닉층의 식은 아래와 같다.



은닉층의 출력은 이제 V의 크기를 가지는 출력층으로 향한다. 이 과정속에서 다시 또 다른 가중치와 곱해지고 bias가 더해지면, 입력 벡터와 동일한 차원의 벡터를 얻음.

출력층에서는 활성함수로 softmax 함수를 사용. 출력을 통한 예측 식은 아래와 같음.

출력 벡터의 각 인덱스에서의 값이 의미하는 것은 0~1사이의 값인데, 이는 각 인덱스의 단어가 다음 단어일 확률을 나타낸다. 실제 값 벡터와 해당 출력 벡터를 가까워지게 하기 위해 손실 함수로 cross-entropy를 사용한다. 그러면서 역전파가 이루어지면서 가중치 행렬들이 학습되는데, 이 과정에서 임베딩 벡터값들도 학습이 된다.

결론적으로, NNLM을 사용하면 충분한 양의 훈련 데이터를 학습한다면, 수많은 문장에서 유사한 목적으로 사용되는 단어들은 결국 유사한 임베딩 벡터값을 얻게 된다는 것이다. 따라서, 훈련 데이터에 없던 단어 시퀀스라 하더라도 다음 단어를 선택할 수 있게 된다.

하지만, 여전히 윈도우라는 개념을 사용하기 때문에 버려지는 단어들이 가지는 문맥 정보는 파악할 수 없다는 단점을 가지고 있다.

**순환 신경망 (RNN)**

이전 신경망들은 전부 은닉층을 지난 값은 출력층 방향으로만 지났다(단방향). => 피드 포워드

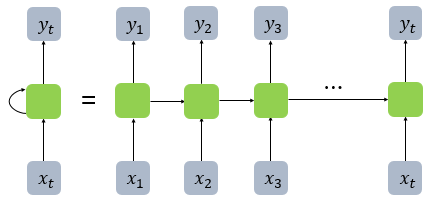
RNN은 은닉층의 노드에서 활성함수를 통해 나온 값을 출력층 방향으로도 보내면서, 다시 은닉층 노드의 다음 계산의 입력으로 보내는 특징을 갖고 있다.

시계이(가) 표시된 사진

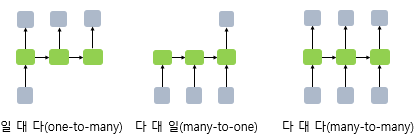
자동 생성된 설명

Cell은 RNN에서 은닉층에서 활성함수를 통해 결과를 내보내는 역할을 하는 노드인데, 이 셀은 이전의 값을 기억하려고 하는 메모리 역할을 수행하므로 메모리 셀, 혹은 RNN 셀이라고도 한다.

이 셀은 바로 이전 시점에서의 은닉층의 셀에서 나온 값을 자신의 입력으로 사용하는 재귀적 활동을 함. => 현재 시점에서의 셀이 갖고 있는 값은 과거의 셀들의 값에 영향을 받았음을 의미함.

(RNN의 구조)

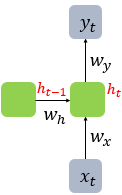
이 구조는 아래처럼 다양한 형태로 나타낼 수도 있다.



일대다 형식은 Image captioning에 많이 사용된다.

단어 시퀀스에 대해 하나의 출력을 하는 모델은 다대일 형식을 띤다. 이는 보통 감성 분류, 스팸 메일 분류 등에 많이 사용된다. => 이번 캡스톤디자인도 욕설 분류이므로 만약 RNN구조를 사용한다면 이 형식의 구조를 사용하게 될 것임.

다대다 형식은 챗봇이나 번역기, 개체명 인식, 품사 태깅과 같은 작업을 포함함.



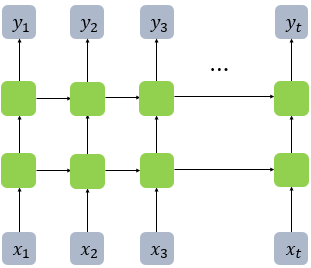
다음 은닉 셀은 결국 두가지 가중치, Wh, Wx를 가지게 된다. 이를 모두 포함해서 결과값인 Yt를 얻는다.

RNN은 keras의 라이브러리를 통해 간단히 불러올 수 있다.

from keras.layers import SimpleRNN

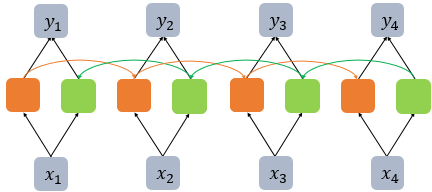
위에서 RNN층은 (batch\_size, timestep, input\_dim) 3차원 텐서를 입력으로 받는다.

리턴 값은 최종 시점의 은닉 상태만 리턴하는 방식이 있고, 모든 은닉 상태를 리턴하는 방식이 있다. 전자는 다대일, 후자는 다대다에 이용된다.

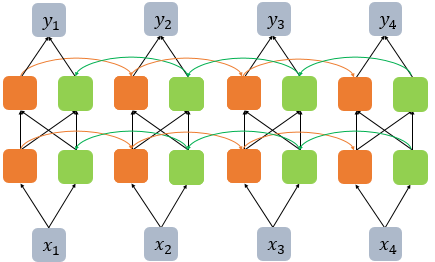
(더 깊은 RNN 구조)

**양방향 순환 신경망 (Bidirectional RNN)**

B-RNN은 특정 시점에서의 출력값을 예측할 때 이전 시점의 데이터뿐만 아니라, 이후 데이터로도 예측할 수 있다는 아이디어에 기반함.

(B-RNN의 구조)

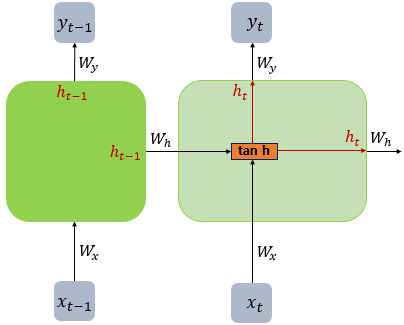
B-RNN은 한 출력값을 예측하기 위해 두 개의 메모리 셀을 이용함. 하나는 앞 시점의 은닉 상태 계산 후 저장, 다른 하나는 뒤 시점의 은닉 상태 계산 후 저장.

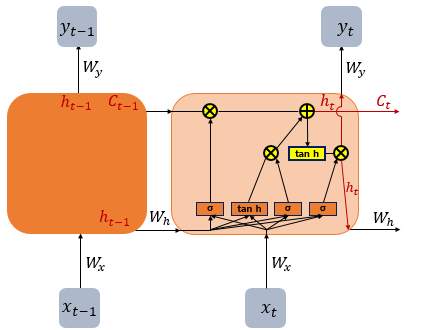
(깊은 B-RNN)

**장단기 메모리(LSTM)**

위의 RNN의 한계를 극복하기 위한 다양한 RNN의 변형 중 하나.

위의 RNN은 비교적 짧은 시퀀스에 대해서만 효과를 보이는 단점이 있다. => 시점이 길어질수록 앞의 정보가 뒤로 충분히 전달되지 못하는 현상 발생.

(기존 RNN 셀의 내부 구조)

(LSTM의 셀의 내부 구조)

LSTM은 은닉층의 셀에 입력 게이트, 망각 게이트, 출력 게이트를 추가해 불필요한 기억을 지우고, 기억해야할 것들을 정함. => CELL STATE 추가

셀 상태는 맨 위의 선(C값이 지나가는 곳)임.

각 게이트는 시그모이드 함수를 지나게 되어 있다.

가운데 하이퍼볼릭탄젠트 함수와 시그모이드 함수를 같이 지나는 곳은 입력 게이트이다.

입력 게이트에서는 기존 RNN연산을 한 뒤, 각각 두 활성함수를 지나게 함.

망각 게이트(삭제 게이트)에서는 현재 시점의 입력 값과 이전 시점의 은닉 상태가 시그모이드 함수를 지나게 된다. 함수를 지나면 0~1값이 나오는데, 이 값이 삭제 과정을 거친 정보의 양. 0에 가까울수록 많이 삭제, 클수록 정보를 온전이 기억.

위의 두 게이트들을 지난 값을 통해 셀 상태(Cell state)를 구하게 됨. 이를 장기 상태라고도 함.

입력 게이트에서 구한 두 값에 원소별 곱을 진행. 이후, 이를 망각 게이트의 결과값과 더함. 이를 현 시점 셀에서의 셀 상태라고 하고, 이 값은 다음 시점의 셀로 넘어감.

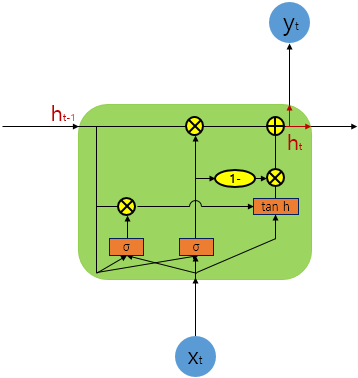
결과적으로, 망각 게이트는 이전 시점의 입력을 얼마나 반영할지를 의미하고, 입력 게이트는 현재 시점의 입력을 얼마나 반영할지를 결정.

출력 게이트는 현재 시점의 x값과 이전 시점의 은닉 상태가 시그모이드 함수를 지난 값. 해당 값은 현재 시점의 은닉 상태를 결정하는데 쓰임.

셀 상태의 값이 하이퍼볼릭탄젠트를 지나 -1~1값이 되고, 이 값이 출력 게이트의 출력 값과 연산되면서 값이 걸러지는 효과가 발생해 은닉 상태가 된다. 이를 단기 상태라고도 함. 이 값은 출력층으로 향한다.

**게이트 순환 유닛(Gated Recureent Unit, GRU)**

GRU는 LSTM의 장기 의존성 문제에 대한 해결책을 유지하면서, 은닉 상태를 업데이트하는 계산을 줄임 => 세 개의 게이트에서 두 개의 게이트로(업데이트 게이트, 리셋 게이트)



보통, 데이터 양이 적을 때는 GRU가 낫고, 많으면 LSTM이 낫다고 알려져 있음.

**워드 임베딩(Word Embedding)**

워드 임베딩은 단어를 벡터로 표현하는 방법으로, 단어를 밀집 표현으로 변환(위에서 잠깐 언급했음).

희소 표현: 벡터 또는 행렬의 값 대부분이 0으로 표현되는 방법, 원-핫 벡터가 예시.

->문제점: 단어의 개수가 늘어나면 차원이 한없이 커진다. = 공간적 낭비

밀집 표현: 벡터의 차원을 단어 개수로 정하지 않고, 사용자가 설정한 값으로 모든 단어 벡터의 차원을 맞춘다. 또한, 0과 1만 쓰지 않고 실수 값을 가지게 됨.

단어를 밀집 벡터 형태로 표현하는 방법을 워드 임베딩이라고 한다. 이 밀집 벡터를 임베딩 벡터라고도 함.

워드 임베딩 방법론: LSA, Word2Vec, FastText, GloVe 등이 있음.

**Word2Vec**

희소 표현은 단어간 유사성을 표현할 수 없다는 단점이 있음.

때문에, 단어의 의미를 다차원 공간에 벡터화 하는 방법을 모색 -> 분산 표현.

분산 표현을 이용해 단어의 유사도를 벡터화하는 작업은 워드 임베딩 작업에 속함. 분산 표현도 밀집 벡터에 속함.

분산 표현은 기본적으로 비슷한 위치에 등장하는 단어들은 비슷한 의미를 가진다라는 가정 하에 만들어진 표현 방법.

ex)강아지는 귀엽다, 예쁘다 등등이 같이 나타나…

이렇게 표현된 벡터들은 벡터의 차원이 단어 집합의 크기일 필요가 없으므로 차원이 상대적으로 줄어듦.

=> 희소 표현이 고차원에 각 차원이 분리된 표현 방법이었다면, 분산 표현은 저차원에 단어의 의미를 여러 차원에다가 분산해 표현한 것. => 단어 간 유사도 계산 가능

W2V에는 CBOW와 Skip-gram 두 가지 방식이 있음.

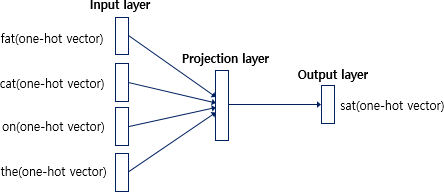
1. CBOW

주변에 있는 단어들을 가지고 중간에 있는 단어들을 예측하는 방법.

테이블이(가) 표시된 사진

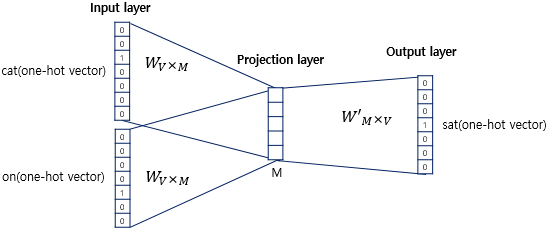
자동 생성된 설명

윈도우 크기를 정해서 앞뒤로 몇 개나 확인할지 정함. 이후 윈도우를 움직여서 학습을 위한 데이터셋을 만든다(슬라이딩 윈도우). 위의 예시는 윈도우 크기가 2(따라서 총 보는 단어의 수는 4개)

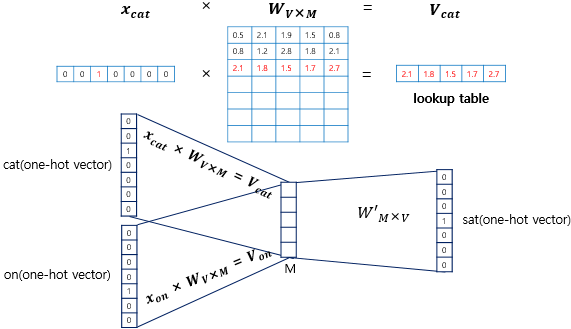


위의 NNLM구조와 유사하다.

입력층에 들어가는 벡터는 임의로 정한 윈도우 크기 범위 안에 있는 주변 단어들의 원-핫 벡터가 들어가게 되고, 출력층에서 예측하고자 하는 중간 단어의 원 핫-벡터가 나온다. W2V의 학습을 위해서 이 출력층의 원-핫 벡터가 필요.

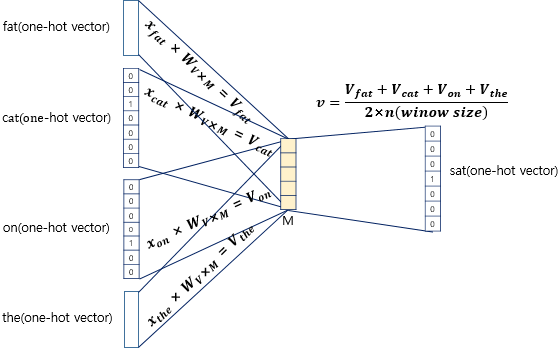
(CBOW)

인공 신경망 훈련 전, 가중치 행렬 W와 W’는 대개 굉장히 작은 랜덤 값으로 초기화 되고 시작한다. CBOW는 주변 단어로 중심 단어를 더 정확히 맞추기 위해 계속해서 이 가중치 행렬들을 학습해 나가는 구조임.



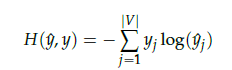
위의 그림은 입력 벡터와 가중치 행렬 W의 곱을 나타낸 그림(왼쪽). 원-핫 벡터는 한 요소만 1이므로 W행렬의 인덱스 1의 값을 그대로 가져오는 것과 같다 => 앞에서 말했던 LOOKUP TABLE

여기서 W의 각 행 벡터는 W2V를 수행한 후의 각 단어의 M차원의 크기를 갖는 임베딩 벡터(M은 투사층의 크기).



그렇게 입력 벡터들에 대해 곱을 다 구하고, 이 벡터들의 평균인 벡터를 구하게 된다. 식은 위 그림에 나와 있음.

이렇게 구해진 평균 벡터는 두번째 가중치 행렬 W’와 곱해지고, 이후 SOFTMAX함수를 지나 0과 1사이의 수가 된다. 이 수로 이루어진 벡터를 스코어 벡터라고 한다. 그리고, 이 스코어 벡터를 실제 원-핫 벡터와의 오차를 줄여가면서…(손실 함수로 CROSS-ENTROPY 사용)

(손실 함수)

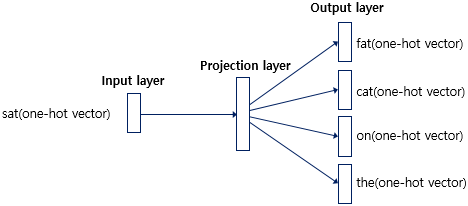
2. Skip-gram

중간에 있는 단어들을 가지고 주변 단어들을 예측하는 방법.

테이블이(가) 표시된 사진

자동 생성된 설명

위의 그림을 도식화하면, 아래와 같음.



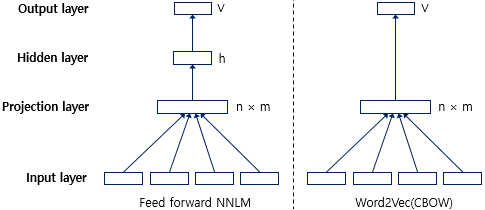
‘fat cat sat on the’의 중심 단어인 ‘sat’을 통해 ‘fat’, ‘cat’, ‘on’, ‘the’를 예측함.

중심 단어 하나만을 가지고 주변 단어를 예측하므로 투사층에서 벡터들의 평균을 구하지 않고 그대로 사용함.

3. 결론

일반적으로 CBOW보다 SKIP-GRAM의 성능이 좋다고 알려져 있음.

**NNLM과 Word2Vec 전격 비교**



NNLM은 단어 간 유사도를 구할 수 있도록 워드 임베딩의 개념을 도입했고, NNLM의 느린 학습 속도와 정확도를 개선해 탄생한 것이 W2V이다.

NNLM은 언어 모델이므로 다음 단어를 예측하지만, W2V는 워드 임베딩 자체가 목적이므로 다음 단어가 아닌 중심 단어를 예측함. => NNLM은 이전 단어들만 참고, W2V는 전후 단어 모두 참고

W2V는 NNLM에 존재하던 활성함수가 존재하는 은닉층을 제거 => 투사층 다음 바로 출력층

W2V는 계층적 SOFTMAX기법과 Negative sampling 기법을 사용해 NNLM보다 학습 속도에서 강점을 가짐.

위의 식은 NNLM의 연산랑이고, 아래의 식은 W2V의 연산량이다. W2V 연상량 수식에서 V가 log(V)로 변환된 것은 앞서 말했던 두가지 기법을 적용했을 때의 일이다.

**글로브(GloVe)**

카운트 기반과 예측 기반을 모두 사용하는 방법론. W2V와 비슷한 성능을 보이니, 두 가지를 다 사용해보고 좋은 쪽을 고르자.

카운트 기반(ex. LSA)는 데이터 전체적인 통계 정보를 고려하긴 하지만, 단어 의미 유추 작업에는 성능이 떨어짐. 예측 기반(ex. W2V)는 단어 간 유추 작업은 뛰어나지만, 임베딩 벡터가 윈도우 크기 내에서만 주변 단어를 고려하기 때문에 데이터 전체적인 통계 정보를 반영하지는 못함.

=> GV는 두 가지 방법론 모두 사용.

단어 동시 등장 행렬은, 행렬의 행과 열을 전체 단어 집합의 단어들로 구성하고, I 단어의 윈도우 크기 내에서 K 단어가 등장한 횟수를 I행 K열에 기재한 행렬.

EX) I like deep learning / I like NLP / I enjoy flying

| **카운트** | **I** | **like** | **enjoy** | **deep** | **learning** | **NLP** | **flying** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| I | 0 | 2 | 1 | 0 | 0 | 0 | 0 |
| like | 2 | 0 | 0 | 1 | 0 | 1 | 0 |
| enjoy | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| deep | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| learning | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| NLP | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| flying | 0 | 0 | 1 | 0 | 0 | 0 | 0 |

(위 세 문장의 단어 동시 등장 행렬)

이 행렬은 전치를 해도 동일하다는 특징을 가짐.

위 행렬으로부터 동시 등장 확률을 구할 수 있음.

동시 등장 확률은 동시 등장 행렬으로부터 특정 단어 i의 전체 등장 횟수를 카운트하고, 특정 단어 i가 등장했을 때 어떤 단어 k가 등장한 횟수를 카운트하여 계산한 조건부 확률임.

즉, 동시 등장 행렬에서 중심 단어 i의 행의 모든 값을 더한 값을 분모로 하고, i행 k열의 값을 분자로 한 값.

GV의 핵심 골자 = 임베딩된 중심 단어와 주변 단어 벡터의 내적이 전체 코퍼스 데이터에서의 동시 등장 확률이 되도록 만드는 것. => 이를 만족하도록 임베딩 벡터를 만드는 것이 목표.

위의 식은 GV의 손실 함수이다.

GV는 파이썬 라이브러리 GV 패키치 설치를 통해 실행해볼 수 있다.

pip install glove\_python

from glove import Corpus, Glove

**FastText**

W2V와 차이점은, W2V는 단어를 쪼개질 수 없는 단위로 생각한다면, FT는 하나의 단어 안에도 여러 단어들이 존재하는 것으로 간주, 즉, 내부 단어를 고려해 학습.

FT에서 각 단어는 글자 단위의 N-GRAM 구성으로 취급.

ex) apple을 3-gram으로 하면 우선 단어 양 끝에 시작과 끝을 의미하는 구분자 <. >를 추가함. 이후, 이 단어는 <ap, app, ppl, ple, le>로 분리되고, 이를 벡터로 만듦. 추가적으로, <apple>도 벡터화 함. 즉, 총 6개의 토큰을 벡터화 함.

실제 사용시에는 n-gram의 n을 범위 지정이 가능함.

ex) n=3~6인 경우, <**ap,** app, ppl, ppl, le>, <**app,** appl, pple, ple>, <**appl,** pple>, ..., <**apple**>

이 내부 단어들을 벡터화함. = 저 단어들에 대해 W2V 수행

이를 통해 얻는 장점? => 데이터 셋만 충분하다면 내부 단어를 통해 모르는 단어(Out of vocabulary, OOV)에 대해서도 다른 단어와의 유사도를 계산할 수 있음.

또, W2V의 경우 등장 빈도 수가 적은 단어에 대해서는 임베딩 정확도가 높지 않다는 단점이 있었는데, FT의 경우 만약 단어가 희귀 단어라도, 그 단어의 N-GRAM이 다른 단어의 N-GRAM과 겹친다면 W2V와 비교하여 비교적 높은 임베딩 벡터 값을 가짐. => 이 말은? 노이즈가 많은 코퍼스(오타가 섞인 코퍼스)에 대해서도 일정 수준 이상의 성능을 보임.

**마무리**

욕설 데이터 셋을 수집하면, 우선적으로 해당 텍스트들에 대하여 전처리가 필요할 것이다(형태소 분석, 워드 임베딩 등).

다양한 워드 임베딩 기법을 사용해 본다(TF-IDF, W2V, FT, GV 등)

임베딩 벡터를 사용해 다양한 언어 모델(LM)에 학습시켜본다(RNN, LSTM, BiLSTM, GRU, NNLM 등)