

Word Embedding

워드 임베딩(Word Embedding)

희소 표현(sparse representation) vs 밀집 표현(dense representation)

- 희소 표현
 - 원-핫 벡터 (0과 1) 등
 - Ex) 강아지 = [0 0 0 0 1 0 0 0 0 0 0 0 ... 중략 ... 0] # 이때 1 뒤의 0의 수는 9995개.
 - 공간적 낭비, 단어의 의미를 표현하지 못함
- 밀집 표현
 - LSA, Word2Vec, FastText, Glove 등
 - 사용자가 설정한 값으로 모든 단어의 벡터 표현의 차원을 맞춤
 - Ex) 강아지 = [0.2 1.8 1.1 -2.1 1.1 2.8 ... 중략 ...] # 이 벡터의 차원은 128
 - 단어를 밀집 벡터(dense vector)의 형태로 표현하는 방법을 워드 임베딩(word embedding)이라고 함

	원-핫 벡터	임베딩 벡터
차원	고차원(단어 집합의 크기)	저차원
다른 표현	희소 벡터의 일종	밀집 벡터의 일종
표현 방법	수동	훈련 데이터로부터 학습함
값의 타입	1과 0	실수

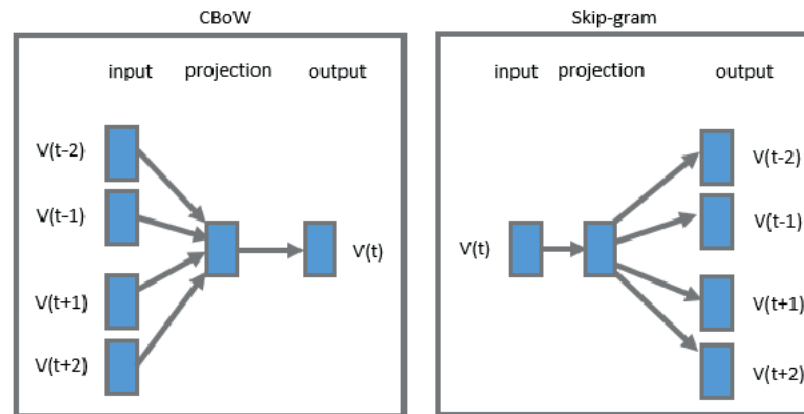
워드투벡터(Word2Vec)

분산 표현(Distributed Representation)

- 희소 표현은 단어 벡터간 유사성 표현 불가 → 의미를 다차원 공간에 벡터화
- '분포 가설' 가정 ('비슷한 문맥에서 등장하는 단어들은 비슷한 의미를 가진다')
- 저차원에 단어의 의미를 여러 차원에다가 분산하여 표현
- 단어 벡터 간 유의미한 유사도 계산 가능

분산 표현의 대표적인 학습 방법인 Word2Vec(2013) 부터 알아보자.

Word2Vec의 학습 방식에는 **CBOW(Continuous Bag of Words)**와 **Skip-Gram** 두 가지 방식이 있음



워드투벡터(Word2Vec)

CBOW(Continuous Bag of Words)

- 주변에 있는 단어들을 입력으로 중간에 있는 단어들을 예측하는 방법

중심 단어
↓
주변 단어

The fat cat sat on the mat

The fat cat sat on the mat

The fat cat sat on the mat

The fat cat sat on the mat

The fat cat sat on the mat

The fat cat sat on the mat

The fat cat sat on the mat

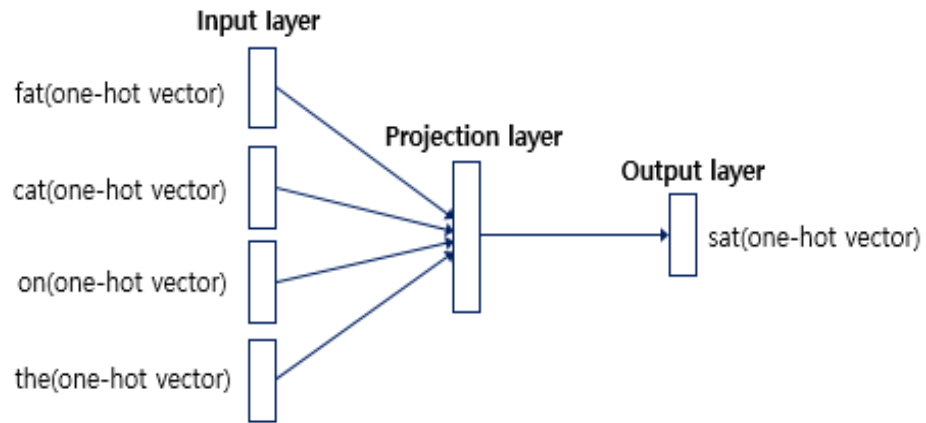
중심 단어	주변 단어
[1, 0, 0, 0, 0, 0, 0]	[0, 1, 0, 0, 0, 0, 0], [0, 0, 1, 0, 0, 0, 0]
[0, 1, 0, 0, 0, 0, 0]	[1, 0, 0, 0, 0, 0, 0], [0, 0, 1, 0, 0, 0, 0], [0, 0, 0, 1, 0, 0, 0]
[0, 0, 1, 0, 0, 0, 0]	[1, 0, 0, 0, 0, 0, 0], [0, 1, 0, 0, 0, 0, 0], [0, 0, 0, 1, 0, 0, 0], [0, 0, 0, 0, 1, 0, 0]
[0, 0, 0, 1, 0, 0, 0]	[0, 1, 0, 0, 0, 0, 0], [0, 0, 1, 0, 0, 0, 0], [0, 0, 0, 0, 1, 0, 0], [0, 0, 0, 0, 0, 1, 0]
[0, 0, 0, 0, 1, 0, 0]	[0, 0, 1, 0, 0, 0, 0], [0, 0, 0, 1, 0, 0, 0], [0, 0, 0, 0, 0, 1, 0], [0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 1, 0]	[0, 0, 0, 1, 0, 0, 0], [0, 0, 0, 0, 1, 0, 0], [0, 0, 0, 0, 0, 0, 1]
[0, 0, 0, 0, 0, 0, 1]	[0, 0, 0, 0, 1, 0, 0], [0, 0, 0, 0, 0, 1, 0]

옆의 예는 윈도우(window) 크기가 2 인 경우

슬라이딩 윈도우(sliding window): 윈도우 크기가 정해지면 윈도우를 옆으로 움직여서 주변 단어와 중심 단어의 선택을 변경해가며 학습을 위한 데이터 셋을 만듦

워드투벡터(Word2Vec)

CBOW(Continuous Bag of Words)



입력층(Input layer)의 입력으로서 앞, 뒤로 사용자가 정한 윈도우 크기 범위 안에 있는 **주변 단어들의 원-핫 벡터**가 들어감

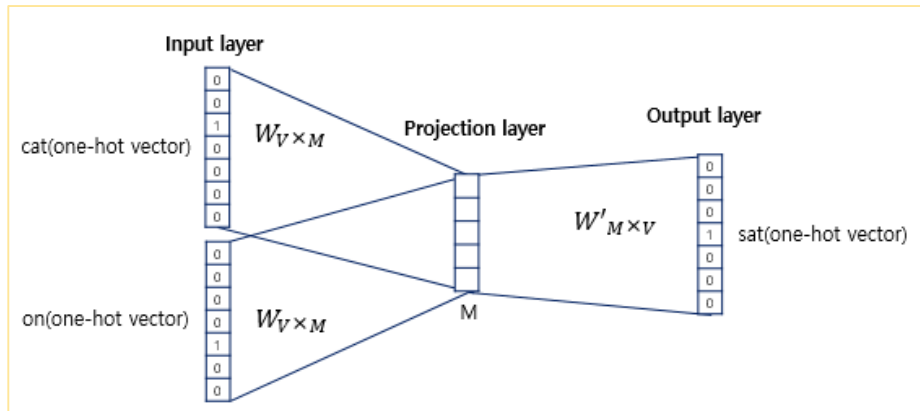
출력층(Output layer)에서 예측하고자 하는 **중간 단어의 원-핫 벡터**가 **레이블**로서 필요

Word2Vec은 은닉층이 다수인 딥러닝 모델이 아니라 은닉층이 1개인 'shallow neural network'

Word2Vec의 은닉층은 활성화 함수가 존재하지 않으며, 룩업 테이블이라는 연산은 담당, **투사층(projection layer)**이라고 부름

워드투벡터(Word2Vec)

CBOW(Continuous Bag of Words)



CBOW에서 투사층의 크기(M)는 임베딩하고 난 벡터의 차원이 됨.

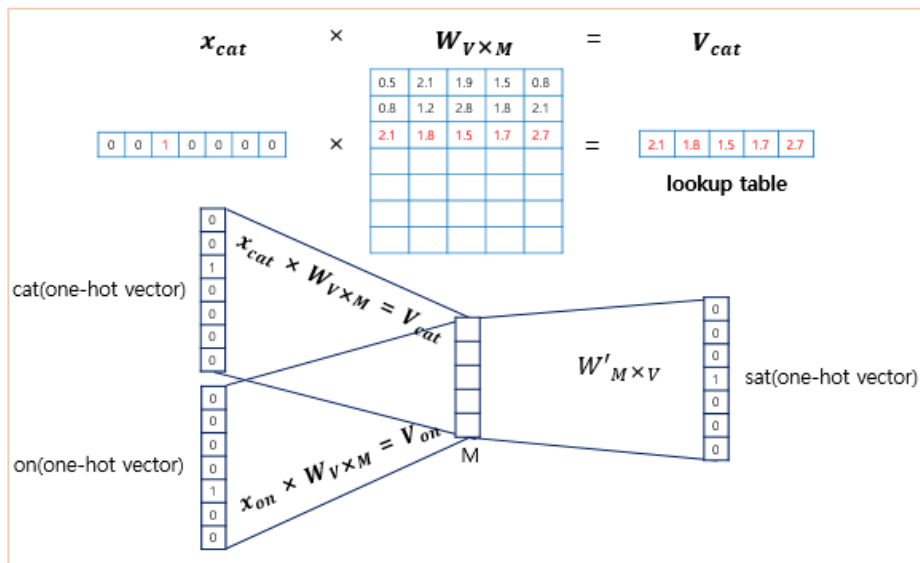
- 그림에서 $M=5$, CBOW를 수행하고나서 얻는 각 단어의 임베딩 벡터의 차원은 5가 됨.

입력층과 투사층 사이의 가중치(W)는 $V \times M$ 행렬이며, 투사층에서 출력층 사이의 가중치(W')는 $M \times V$ 행렬임(여기서 V 는 단어 집합 크기)

- 그림에서 $V=7$, W는 7×5 행렬, W'는 5×7 행렬

인공 신경망 훈련 전에 가중치 행렬 W와 W'는 랜덤 값

- CBOW는 주변 단어로 중심 단어를 더 정확히 맞추기 위해 계속해서 이 **W와 W'**를 학습해가는 구조

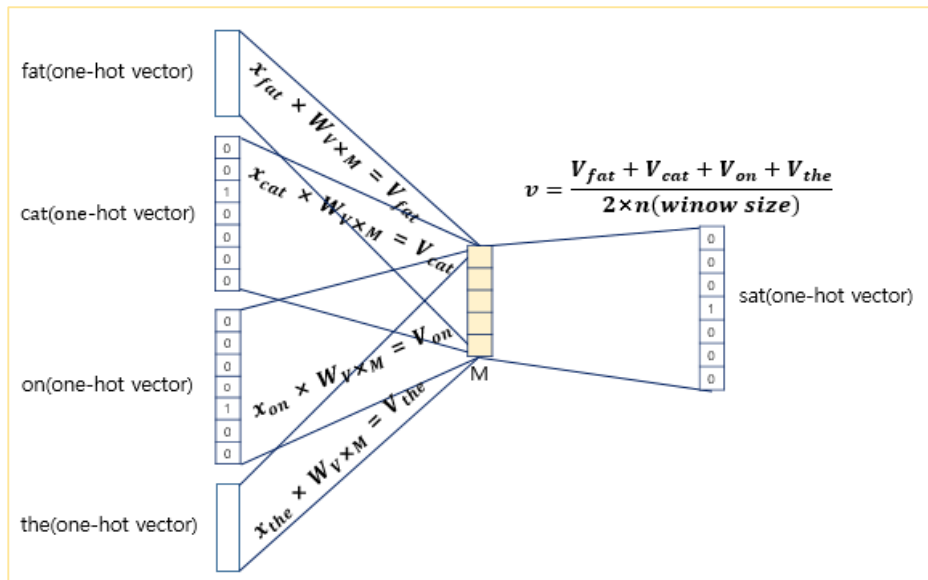


입력 벡터 x와 가중치 W 행렬의 곱은 사실 W행렬의 i번째 행을 그대로 읽어오는 것과(lookup) 동일

lookup해온 W의 각 행벡터가 Word2Vec 학습 후에는 각 단어의 M차원의 **임베딩 벡터**로 간주됨 = V_i (단어집합 크기 V와 다름!)

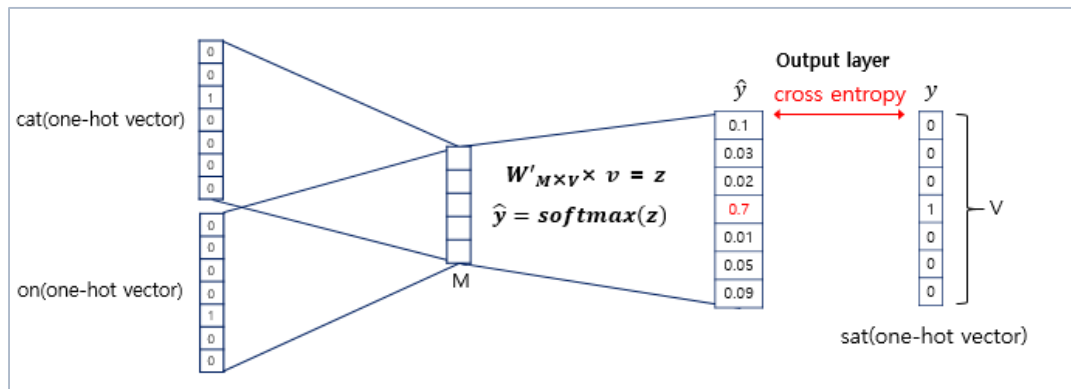
워드투벡터(Word2Vec)

CBOW(Continuous Bag of Words)



주변 단어의 원-핫 벡터(x)에 대해 가중치 W가 곱해서 생겨진 '결과 벡터들'(V_i)은 투사층에서 만나 **평균인 벡터**를 구하게 됨.

투사층에서 벡터의 평균을 구하는 부분은 CBOW가 Skip-Gram과 다른 차이점이기도 함



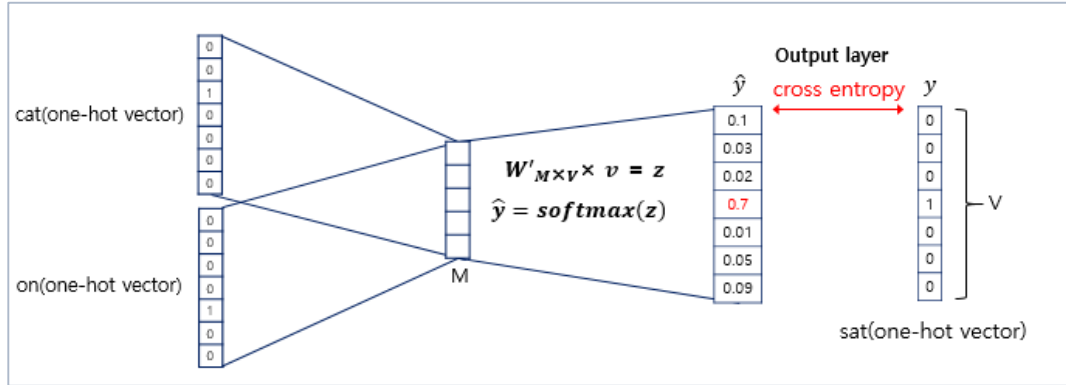
구해진 평균 벡터(v)는 두번째 가중치 행렬 W' 와 곱해지며, 곱셈의 결과(z)로는 원-핫 벡터들과 차원이 V로 동일한 벡터가 나옴

이 벡터(z)에 CBOW는 **소프트맥스(softmax)** 함수를 지나고, 벡터의 각 원소들의 값은 **0과 1사이의 실수**를 가짐(총 합 1)

- 이는 다중 클래스 분류 문제를 위한 일종의 스코어 벡터(score vector)
- **스코어 벡터(\hat{y})의 j번째 인덱스가 가진 값은 j번째 단어가 중심 단어일 확률임**
- 이 스코어 벡터의 값은 **레이블(y)**에 해당하는 벡터인 중심 단어 원-핫 벡터의 값에 **가까워져야 함**

워드투벡터(Word2Vec)

CBOW(Continuous Bag of Words)



이 두 벡터값(\hat{y} , y)의 오차를 줄이기위해 CBOW는 손실 함수(loss function)로 크로스 엔트로피(cross-entropy) 함수 사용

$$\text{cost}(\hat{y}, y) = - \sum_{j=1}^V y_j \log(\hat{y}_j)$$

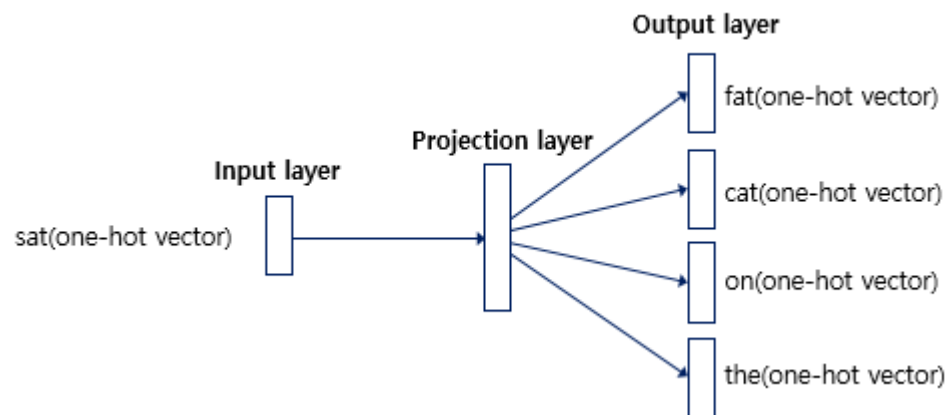
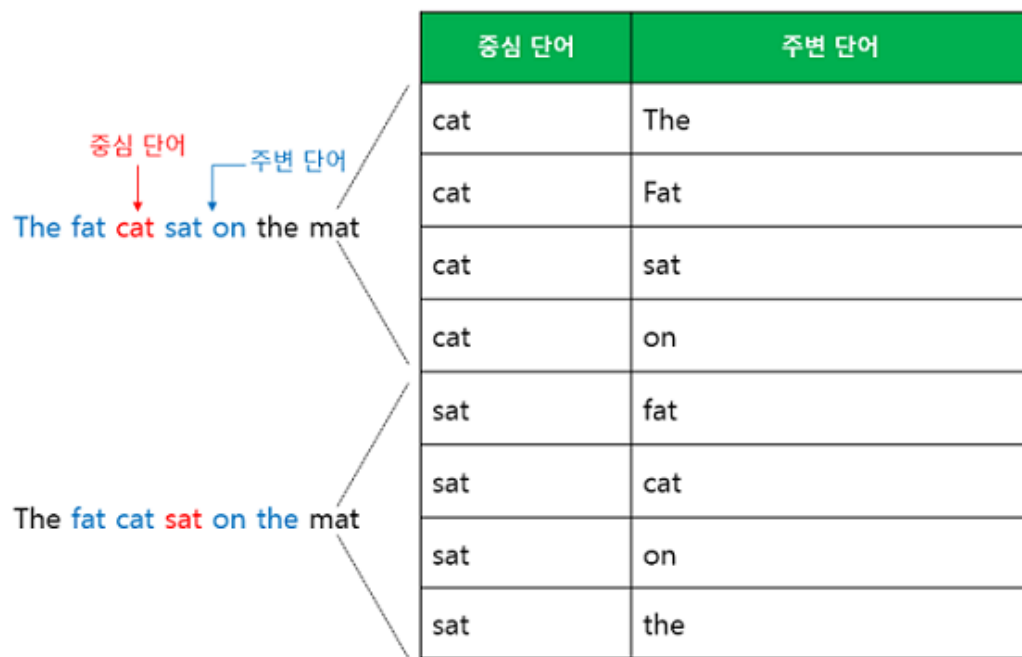
역전파(Back Propagation)를 수행하면 W와 W'가 학습됨

학습이 다 되었다면 M차원의 크기를 갖는 W의 행렬의 행을 각 단어의 임베딩 벡터로 사용하거나 W와 W' 행렬 두 가지 모두를 가지고 임베딩 벡터를 사용하기도 함.

워드투벡터(Word2Vec)

Skip-gram

- 중심에 있는 단어를 입력으로 주변에 있는 단어들을 예측하는 방법
- 나머지 방식은 CBOW와 동일



워드투벡터(Word2Vec)

Skip-gram

보통 한 단어에 대해 여러 학습 기회를 가지는 Skip-gram의 성능이 더 좋다고 함

The fat cat sat on the table, window size=2

CBOW						
Input				Output		
Fat, cat				The		
The, cat, sat				Fat		
The, fat, sat, on				Cat		
fat,cat,on,the				Sat		
Cat,sat,the,table				On		
Sat, on, table				The		
On,the				table		
the	fat	cat	sat	on	the	table
1	1	1	1	1	1	1

SKIP-GRAM						
Input				Output		
The				Fat,cat		
Fat				The,cat,sat		
Cat				The,fat,sat,on		
Sat				Fat,cat,on,the		
On				Cat,sat,the,table		
The				Sat,on,table		
table				On,the		
the	fat	cat	sat	on	the	table
2	3	4	4	4	3	2

워드투벡터(Word2Vec)

만약 단어 집합의 크기가 수만 이상에 달한다면 Word2Vec은 학습하기에 무거운 모델이 됨.

- Word2Vec은 역전파 과정에서 **모든 단어의 임베딩 벡터값의 업데이트를 수행**

- 현재 집중하고 있는 중심 단어와 주변 단어가 '강아지'와 '고양이', '귀여운'과 같은 단어라면, 사실 이 단어들과 별 연관 관계가 없는 '돈가스'나 '컴퓨터'와 같은 수많은 단어의 임베딩 벡터값까지 업데이트하는 것은 **비효율적**임.

네거티브 샘플링(negative sampling)은 Word2Vec이 학습 과정에서 전체 단어 집합이 아니라 **일부 단어 집합에만 집중**할 수 있도록 하는 방법

- 하나의 중심 단어에 대해서 전체 단어 집합보다 훨씬 작은 단어 집합을 만들어 놓고 마지막 단계를 이진 분류 문제로 변환

- 주변 단어들을 **긍정(positive)**, 랜덤으로 샘플링 된 단어들을 **부정(negative)**으로 레이블링한다면 **이진 분류 문제**를 위한 데이터셋이 됨.

- 이는 기존의 단어 집합의 크기만큼의 선택지를 두고 다중 클래스 분류 문제를 풀던 Word2Vec보다 훨씬 연산량에서 효율적임.

워드투벡터(Word2Vec)

Skip-gram with negative sampling(SGNS)

- SGNS는 중심 단어와 주변 단어가 모두 입력이 되고, 두 단어가 실제 윈도우 크기 내에 존재하는 이웃 관계인지 그 확률을 예측

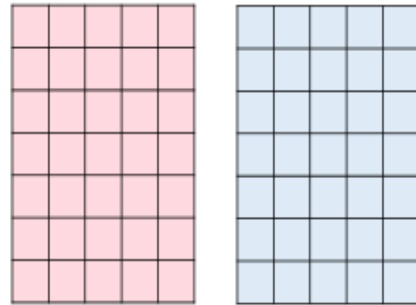


워드투벡터(Word2Vec)

Skip-gram with negative sampling(SGNS)

입력1	입력2	레이블
cat	The	1
cat	fat	1
cat	pizza	0
cat	computer	0
cat	sat	1
cat	on	1
cat	cute	1
cat	mighty	0
...

Embedding layer Embedding layer

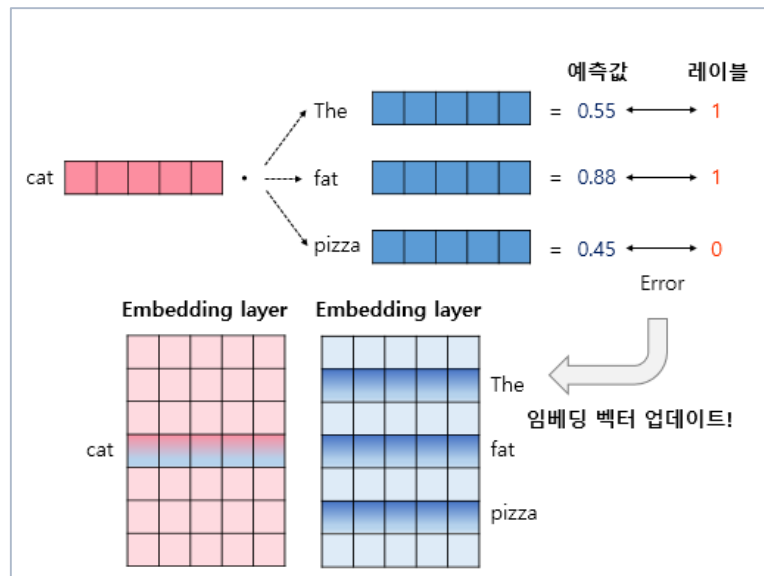
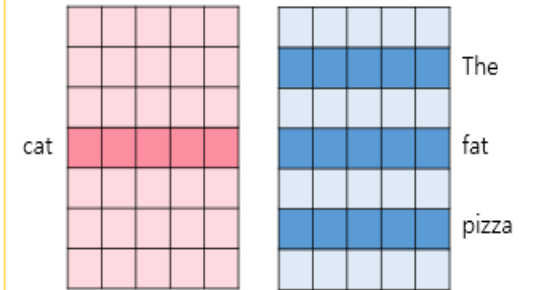


두 테이블 중

하나는 입력 1인 중심 단어의 테이블 룩업을 위한 임베딩 테이블,
하나는 입력 2인 주변 단어의 테이블 룩업을 위한 임베딩 테이블

각 단어는 각 임베딩 테이블을 테이블 룩업하여 임베딩 벡터로 변환

Embedding layer Embedding layer



중심 단어와 주변 단어의 내적값을 이 모델의 예측값으로 하고,
레이블과의 오차로부터 역전파하여 중심 단어와 주변 단어의 임베딩
벡터값을 업데이트

워드투벡터(Word2Vec)

Skip-gram with negative sampling(SGNS) - Keras 이용

```
from tensorflow.keras.preprocessing.sequence import skipgrams
```

```
skip_grams = [skipgrams(sample, vocabulary_size=vocab_size, window_size=10) for sample in encoded]
```

```
embedding_dim = 100
```

```
# 중심 단어를 위한 임베딩 테이블
```

```
w_inputs = Input(shape=(1, ), dtype='int32')
```

```
word_embedding = Embedding(vocab_size, embedding_dim)(w_inputs)
```

```
# 주변 단어를 위한 임베딩 테이블
```

```
c_inputs = Input(shape=(1, ), dtype='int32')
```

```
context_embedding = Embedding(vocab_size, embedding_dim)(c_inputs)
```

```
dot_product = Dot(axes=2)([word_embedding, context_embedding])
```

```
dot_product = Reshape((1, ), input_shape=(1, 1))(dot_product)
```

```
output = Activation('sigmoid')(dot_product)
```

```
model = Model(inputs=[w_inputs, c_inputs], outputs=output)
```

```
model.summary()
```

```
model.compile(loss='binary_crossentropy', optimizer='adam')
```

```
for epoch in range(1, 6):
```

```
    loss = 0
```

```
    for _, elem in enumerate(skip_grams):
```

```
        first_elem = np.array(list(zip(*elem[0]))[0], dtype='int32')
```

```
        second_elem = np.array(list(zip(*elem[0]))[1], dtype='int32')
```

```
        labels = np.array(elem[1], dtype='int32')
```

```
        X = [first_elem, second_elem]
```

```
        Y = labels
```

```
        loss += model.train_on_batch(X,Y)
```

```
    print('Epoch :',epoch, 'Loss :',loss)
```

```
Epoch: 1 Loss: 4339.997158139944
```

```
Epoch: 2 Loss: 3549.69356325455
```

```
Epoch: 3 Loss: 3295.072506020777
```

```
Epoch: 4 Loss: 3038.1063768607564
```

```
Epoch: 5 Loss: 2790.9479411702487
```

글로브(GloVe)

글로브(Global Vectors for Word Representation) (2014, Stanford)

: 카운트 기반, 예측기반을 모두 사용하는 방법론

카운트 기반

- LSA(Latent Semantic Analysis): DTM이나 TF-IDF 행렬과 같이 각 문서에서의 각 단어의 빈도수를 카운트 한 행렬이라는 전체적인 통계 정보를 입력으로 받아 차원을 축소(Truncated SVD)하여 잠재된 의미를 끌어내는 방법론
- 단어 의미의 유추 작업(Analogy task)에 성능이 떨어짐

예측 기반

- Word2Vec: 실제값과 예측값에 대한 오차를 손실 함수를 통해 줄여나가며 학습
- 임베딩 벡터가 윈도우 크기 내에서만 주변 단어를 고려하기 때문에 코퍼스의 전체적인 통계 정보를 반영하지 못함

글로브(GloVe)

윈도우 기반 동시 등장 행렬(window based co-occurrence matrix)

- 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

글로브(GloVe)

동시 등장 확률(co-occurrence probability)

동시 등장 확률과 크기 관계 비(ratio)	k=solid	k=gas	k=water	k=fasion
P(k l ice)	0.00019	0.000066	0.003	0.000017
P(k l steam)	0.000022	0.00078	0.0022	0.000018
P(k l ice) / P(k l steam)	8.9	0.085	1.36	0.96

GloVe의 아이디어를 한 줄로 요약하면 '임베딩 된 중심 단어와 주변 단어 벡터의 내적이 전체 코퍼스에서의 동시 등장 확률(의 로그값)이 되도록 만드는 것'

$$\text{dot product}(w_i \tilde{w}_k) \approx \log P(k | i) = \log P_{ik}$$

→ 손실 함수 설계

- X : 동시 등장 행렬(Co-occurrence Matrix)
- X_{ij} : 중심 단어 i가 등장했을 때 윈도우 내 주변 단어 j가 등장하는 횟수
- $X_i: \sum_j X_{ij}$: 동시 등장 행렬에서 i행의 값을 모두 더한 값
- $P_{ik}: P(k | i) = \frac{X_{ik}}{X_i}$: 중심 단어 i가 등장했을 때 윈도우 내 주변 단어 k가 등장할 확률
Ex) P(solid l ice) = 단어 ice가 등장했을 때 단어 solid가 등장할 확률
- $\frac{P_{ik}}{P_{jk}}$: P_{ik} 를 P_{jk} 로 나눠준 값
Ex) P(solid l ice) / P(solid l steam) = 8.9
- w_i : 중심 단어 i의 임베딩 벡터
- \tilde{w}_k : 주변 단어 k의 임베딩 벡터

글로브(GloVe)

손실함수

$$F(w_i, w_j, \tilde{w}_k) = \frac{P_{ik}}{P_{jk}}$$

두 개의 중심단어가 주어지고 하나의 주변단어가 주어질 때 어떠한 함수 F를 통해 동시 발생 확률 비율을 도출

$$F(w_i - w_j, \tilde{w}_k) = \frac{P_{ik}}{P_{jk}}$$

벡터 간 관계를 따져보기 위해 w_i 와 w_j 를 뺀 벡터에 w_k 를 내적함.
(빼는 이유는 추후 준동형(Homomorphism)이 성립 되어 손쉽게 처리할 수 있기 때문)

$$F((w_i - w_j)^T \tilde{w}_k) = \frac{P_{ik}}{P_{jk}}$$

$$F(a + b) = F(a)F(b), \forall a, b \in \mathbb{R} \quad \text{준동형(Homomorphism)식 (F는 준동형 식이라는 필수 조건 설정)}$$

$$F(v_1^T v_2 + v_3^T v_4) = F(v_1^T v_2)F(v_3^T v_4), \forall v_1, v_2, v_3, v_4 \in V \quad \text{준동형(Homomorphism)식 - 벡터 값}$$

$$F(v_1^T v_2 - v_3^T v_4) = \frac{F(v_1^T v_2)}{F(v_3^T v_4)}, \forall v_1, v_2, v_3, v_4 \in V \quad \text{준동형(Homomorphism)식 - 벡터 값 뺄셈}$$

$$F((w_i - w_j)^T \tilde{w}_k) = \frac{F(w_i^T \tilde{w}_k)}{F(w_j^T \tilde{w}_k)}$$

준동형(Homomorphism)식을 GloVe식에 적용

글로브(GloVe)

손실함수

$$\frac{P_{ik}}{P_{jk}} = \frac{F(w_i^T \tilde{w}_k)}{F(w_j^T \tilde{w}_k)} \quad \text{이전 식을 좌변에 대입}$$

$$F(w_i^T \tilde{w}_k) - F(w_j^T \tilde{w}_k) = \frac{F(w_i^T \tilde{w}_k)}{F(w_j^T \tilde{w}_k)} \quad \text{좌변을 풀어 쓰면, 뺄셈에 대한 준동형식과 일치}$$

$$\exp(w_i^T \tilde{w}_k) - \exp(w_j^T \tilde{w}_k) = \frac{\exp(w_i^T \tilde{w}_k)}{\exp(w_j^T \tilde{w}_k)} \quad \text{이러한 식을 만족하는 함수로 지수 함수(exponential function)이 있음!}$$

$$\exp(w_i^T \tilde{w}_k) = P_{ik} = \frac{X_{ik}}{X_i}$$

다음 식 도출 가능

$$w_i^T \tilde{w}_k = \log P_{ik} = \log \left(\frac{X_{ik}}{X_i} \right) = \log X_{ik} - \log X_i$$

$$w_i^T \tilde{w}_k + b_i + \tilde{b}_k = \log X_{ik}$$

$\log X_i$ 를 b_i 와 b_k 라는 편향 상수항으로 대체

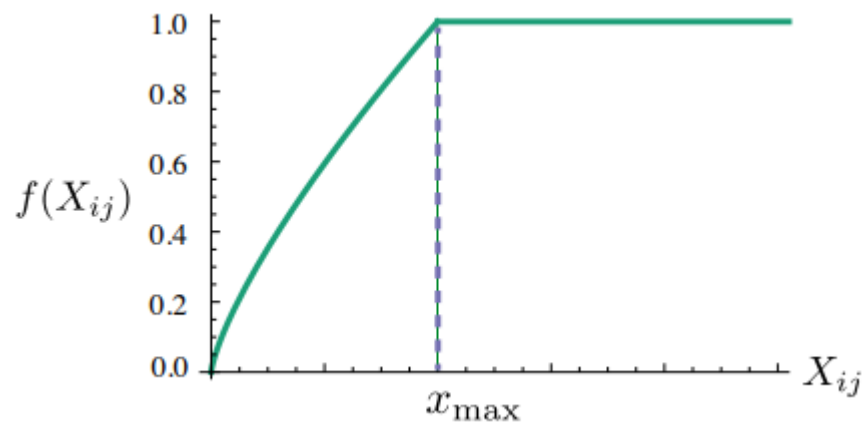
좌변과 우변의 차를 손실 함수로 정의

글로브(GloVe)

손실함수

$$Loss\ function = \sum_{m,n=1}^V (w_m^T \tilde{w}_n + b_m + \tilde{b}_n - \log X_{mn})^2$$

단, X_{ik} 가 0이 될 수 있음 \rightarrow 가중치 함수 $f(X_{ij})$ 를 도입



$$f(x) = \min(1, (x/x_{max})^{3/4})$$

$$Loss\ function = \sum_{m,n=1}^V f(X_{mn})(w_m^T \tilde{w}_n + b_m + \tilde{b}_n - \log X_{mn})^2$$

최종적인 일반화된 손실 함수

글로브(GloVe)

```
from glove import Corpus, Glove

corpus = Corpus()

# 훈련 데이터로부터 GloVe에서 사용할 동시 등장 행렬 생성
corpus.fit(result, window=5)
glove = Glove(no_components=100, learning_rate=0.05)

# 학습에 이용할 쓰레드의 개수는 4로 설정, 에포크는 20.
glove.fit(corpus.matrix, epochs=20, no_threads=4, verbose=True)
glove.add_dictionary(corpus.dictionary)
```

패스트텍스트(FastText)

패스트텍스트(FastText) (2015, facebook)

Word2Vec의 확장, 단 **하나의 단어에도 여러 단어들이 존재**하는 것으로 간주. 즉, **서브워드(subword)**를 고려하여 학습

서브워드(subword)

- 각 단어를 글자 단위 n-gram의 구성을 취급
- 단 시작과 끝인 <, >, 그리고 기존 단어 토큰 <word>를 추가



- 기본 값으로 최소=3, 최대=6으로 설정 되어 있음.
- 이 때, 서브워드를 벡터화하고,
- 단어(ex. apple)의 벡터값은 벡터값들의 총 합으로 구성함

n = 3 ~ 6인 경우

<ap, app, ppl, ppl, le>, <app, appl, pple, ple>, <appl, pple>, ..., <apple>

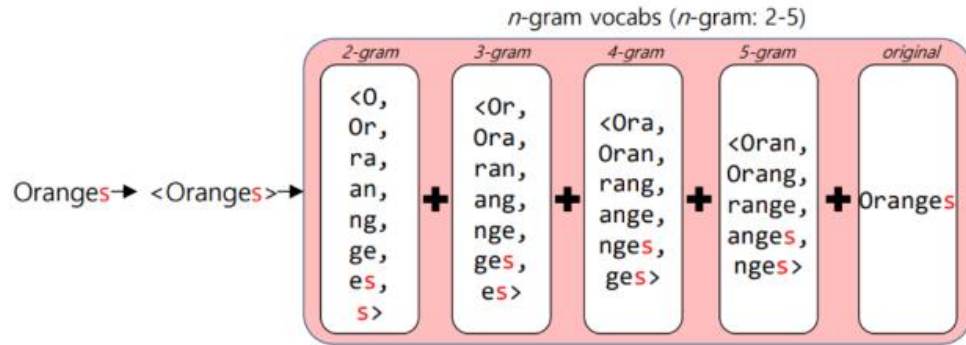
apple = <ap + app + ppl + ppl + le> + <app + appl + pple + ple> + <appl + pple> + , ..., +<apple>



패스트텍스트(FastText)

모르는 단어(out of vocabulary, OOV)에 대한 대응

- FastText의 인공 신경망을 학습한 후에는 데이터 셋의 모든 단어의 각 n-gram에 대해서 워드 임베딩이 됨.
- subword를 통해 모르는 단어(OOV)에 대해서도 다른 단어와의 유사도를 계산할 수 있음.
- Ex. 'birthplace'는 학습되지 않고, 'birth'와 'place'는 되었다면, FastText는 'birthplace' 벡터를 얻을 수 있음.



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

Word2Vec

```
model.wv.most_similar("electrofishing")
```

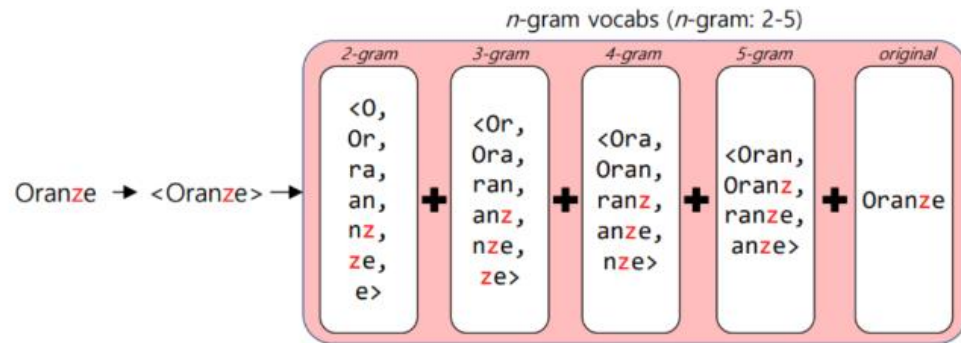
FastText

```
[('electrolux', 0.7934642434120178), ('electrolyte', 0.78279709815979), ('electro', 0.779127836227417), ('electric', 0.7753111720085144), ('airbus', 0.7648627758026123), ('fukushima', 0.7612422704696655), ('electrochemical', 0.7611693143844604), ('gastric', 0.7483425140380859), ('electroshock', 0.7477173805236816), ('overfishing', 0.7435552477836609)]
```

패스트텍스트(FastText)

빈도 수가 적었던 단어(rare word)에 대한 대응

- Word2Vec은 등장 빈도가 적은 단어에 대해 임베딩 정확도가 낮음.
- 하지만 FastText는 그 단어의 n-gram이 다른 단어의 n-gram과 겹치면 정확도 상승
- 또한 노이즈(오타 등)가 많은 코퍼스에서 강점을 가짐. (ex. oranze)



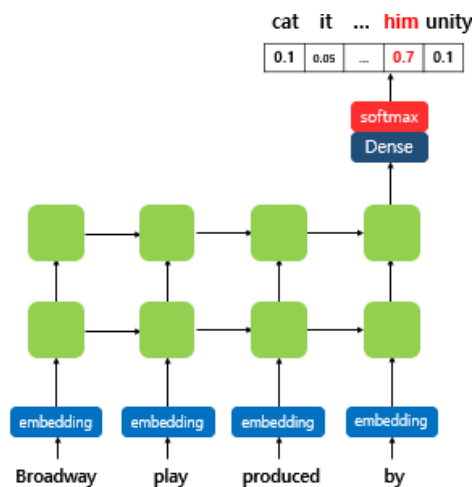
엘모(ELMo)

ELMo(Embeddings from Language Model)

- 문맥을 반영한 워드 임베딩(Contextualized Word Embedding)
- Ex. Bank는 Bank Account(은행 계좌)와 River Bank(강둑)에서 전혀 다른 의미를 가짐. 이 때 Word2Vec이나 GloVe 등으로 표현된 임베딩 벡터들은 이를 제대로 반영하지 못함.

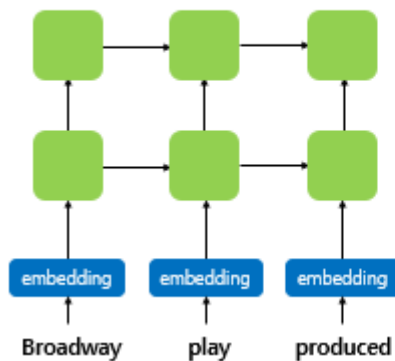
biLM(Bidirectional Language Model)

- ELMo는 순방향 RNN 뿐만 아니라, 위의 그림과는 반대 방향으로 문장을 스캔하는 역방향 RNN 또한 활용. 즉, ELMo는 양쪽 방향의 언어 모델을 둘 다 학습하여 활용

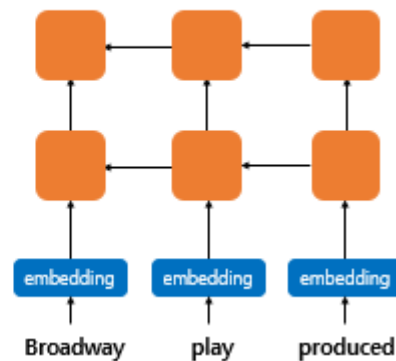


순방향 RNN

순방향 언어 모델
(Forward Language Model)



역방향 언어 모델
(Backward Language Model)



biLM, 다층 구조(Multi-layer)(은닉층이 최소 2개 이상)를 전제로 함

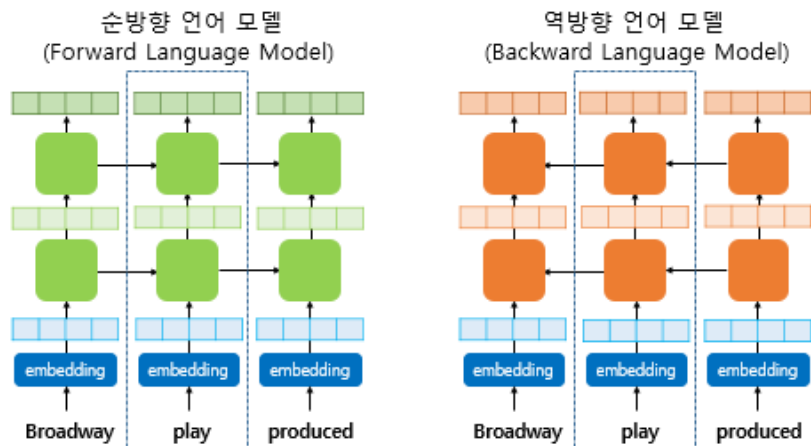
엘모(ELMo)

biLM(Bidirectional Language Model)

- biLM의 각 시점의 입력이 되는 단어 벡터는 **합성곱 신경망**을 이용한 **문자 임베딩(character embedding)**을 통해 얻은 단어 벡터임.
 - 문자 임베딩은 마치 서브단어(subword)의 정보를 참고하는 것처럼 문맥과 상관없이 dog란 단어와 doggy란 단어의 연관성을 찾아낼 수 있음.
- 양방향 RNN vs. ELMo의 biLM
 - 양방향 RNN은 순방향 RNN의 은닉 상태와 역방향의 RNN의 은닉 상태를 연결(concatenate)하여 다음층의 입력으로 사용
 - 반면, **biLM의 순방향 언어모델과 역방향 언어모델이라는 두 개의 언어 모델을 별개의 모델로 보고 학습**

엘모(ELMo)

biLM의 활용



$$\begin{aligned} & \begin{matrix} \text{Green vector} \\ \text{Orange vector} \\ \text{Blue vector} \end{matrix} \times \begin{matrix} S_1 \\ S_2 \\ S_3 \end{matrix} \\ & + \\ & \begin{matrix} \text{Green vector} \\ \text{Orange vector} \\ \text{Blue vector} \end{matrix} \times \begin{matrix} S_2 \\ S_1 \\ S_3 \end{matrix} \\ & + \\ & \begin{matrix} \text{Blue vector} \\ \text{Orange vector} \\ \text{Green vector} \end{matrix} \times \begin{matrix} S_3 \\ S_2 \\ S_1 \end{matrix} \\ & = \text{Red vector} \\ & \gamma \times \text{Red vector} = \text{Red vector} \end{aligned}$$

play라는 단어를 임베딩 하기 위해 점선 사각형 내부의 각 층의 결과값을 재료로 사용 즉, 해당 시점(time step)의 BiLM의 각 층의 출력값을 가져옴

그리고 **순방향 언어 모델과 역방향 언어 모델의 각 층의 출력값을 연결**
여기서 각 층의 출력값이란 첫번째는 임베딩 층을 말하며, 나머지 층은 각 층의 은닉 상태를 말함.

ELMo의 직관적인 아이디어는 **각 층의 출력값이 가진 정보는 전부 서로 다른 종류의 정보를 갖고 있을 것이므로, 이들을 모두 활용한다는 점**

- 1) 각 층의 출력값을 연결(concatenate)
- 2) 각 층의 출력값 별로 가중치를 줌
- 3) 각 층의 출력값을 모두 더함
- 4) 벡터의 크기를 결정하는 스칼라 매개변수를 곱함
→ 완성된 벡터를 **ELMo 표현(representation)**이라고 함

이 ELMo표현을 입력으로 사용하여 수행하고 싶은 텍스트 분류, 질의 응답 시스템 등의 자연어 처리 작업

엘모(ELMo)

ELMo 표현의 사용

ELMo 표현은 기존의 임베딩 벡터와 함께 사용할 수 있음.

우선 텍스트 분류 작업을 위해서 GloVe와 같은 **기존의 방법론을 사용한 임베딩 벡터를 준비** + **준비된 ELMo 표현을 GloVe 임베딩 벡터와 연결(concatenate)**해서 입력으로 사용

이때 biLM의 가중치는 고정시키고, 위에서 사용한 s_1, s_2, s_3 와 γ 는 훈련 과정에서 학습됨.

```
elmo = hub.Module("https://tfhub.dev/google/elmo/1", trainable=True)
```

```
# 텐서플로우 허브로부터 ELMo를 다운로드
```

```
def ELMoEmbedding(x):
```

```
    return elmo(tf.squeeze(tf.cast(x, tf.string)), as_dict=True, signature="default")["default"]
```

```
# 데이터의 이동이 케라스 → 텐서플로우 → 케라스가 되도록 하는 함수
```

```
input_text = Input(shape=(1, ), dtype=tf.string)
```

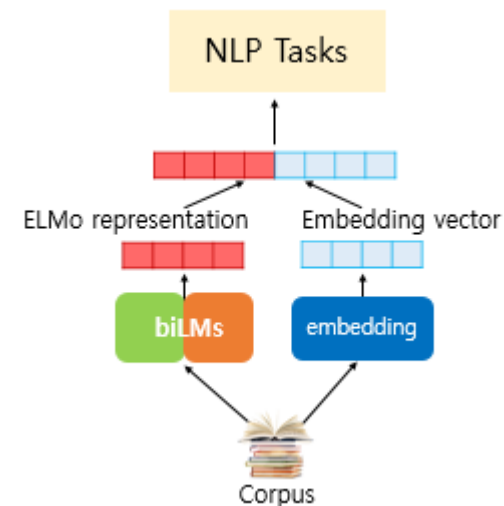
```
embedding_layer = Lambda(ELMoEmbedding, output_shape=(1024, ))(input_text)
```

```
hidden_layer = Dense(256, activation='relu')(embedding_layer)
```

```
output_layer = Dense(1, activation='sigmoid')(hidden_layer)
```

```
model = Model(inputs=[input_text], outputs=output_layer)
```

```
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
```



참고자료

- 딥 러닝을 이용한 자연어 처리 입문(<https://wikidocs.net/book/2155>)
- 자모 단위의 한국어 FastText 이해와 실습
([https://museonghwang.github.io/nlp\(natural%20language%20processing\)/2023/02/10/nlp-kor-fasttext/](https://museonghwang.github.io/nlp(natural%20language%20processing)/2023/02/10/nlp-kor-fasttext/))