[4주차] 워드임베딩

Ch.9 워드 임베딩 (Word Embedding)

01 워드 임베딩

워드 임베딩이란?

텍스트를 컴퓨터가 이해하고, 효율적으로 처리하기 위해서는 컴퓨터가 이해할 수 있도록 텍스트를 적절한 숫자로 변환해야 하는데, **단어를 수치화 하기 위해 단어를 인공 신경망 학습을 통해 벡터화 하는 방법**

1) 희소표현(Sparse Representation)

ex) **원-핫 벡터(one-hot encoding)**

: 원-핫 인코딩으로 표현하고자 하는 단어의 인덱스의 값만 1이고, 나머지 인덱스에는 전부 0으로 표현되는 벡터 표현법
⇒ 벡터 또는 행렬의 값이 대부분 0으로 표현됨 (**희소 표현**_sparse representation)
⇒ 원핫 벡터는 **희소벡터**

희소벡터의 문제점
: 단어의 개수가 늘어나면 벡터의 차원이 한없이 커짐(벡터의 차원 = 단어 집합의 크기) ⇒ 공간 낭비

> 📢 강아지 = [0 0 0 0 1 0 0 0 0 ... 중략 ... 0] # 이때 1 뒤의 0의 수는 9995개, 차원은 10000

2) 밀집 표현(Dense Representation)

- 희소표현과 반대되는 표현으로, 벡터의 차원을 단어 집합의 크기로 상정하지 않고 사용자가 설정한 값으로 모든 단어의 벡터 표현의 차원을 맞춤

사용자가 밀집 표현의 차원을 128로 설정한다면, 모든 단어의 벡터 표현의 차원은 128로 바뀌면서 모든 값이 실수

📢 강아지 = [0.2 1.8 1.1 -2.1 1.1 2.8 ... 중략 ...] # 이 벡터의 차원은 128

⇒ 벡터의 차원이 조밀해져, **밀집 벡터**라고 함

3) 워드 임베딩(Word Embedding)

워드 임베딩 : 단어를 밀집벡터의 형태로 표현하는 방법

임베딩 벡터 : 위의 밀집 벡터 (워드 임베딩 과정을 통해 나온 결과라 하여)

워드 임베딩 방법론 : LSA, Word2Vec, FastText, Glove 등

케라스에서 제공하는 도구인 `Embedding()`

- 워드 임베딩 방법론인 LSA, Word2Vec, FastText, Glove 등을 사용하지는 않지만,
  단어를 랜덤한 값을 가지는 밀집 벡터로 변환한 뒤에, 인공 신경망의 가중치를 학습하는 것과 같은 방식으로 단어 벡터를 학습

02 글로브(Global Vectors for Word Representation, GloVe)

: **카운트 기반**과 **예측 기반**을 모두 사용하는 방법론

현재까지의 연구에 따르면 단정적으로 Word2Vec와 GloVe 중에서 어떤 것이 더 뛰어나다고 말할 수는 없고,
이 두 가지 전부를 사용해보고 성능이 더 좋은 것을 사용하는 것이 바람직함

1) 기존 방법론에 대한 비판

LSA(DTM, TF-IDF 행렬**)**

: 각 문서에서의 **각 단어의 빈도수를 카운트** 한 행렬이라는 전체적인 통계 정보를 입력으로 받아 차원을 축소(Truncated SVD)하여 잠재된 의미를 끌어내는 방법론

장점: 카운트 기반으로 코퍼스의 **전체적인 통계 정보**를 고려함

단점: 왕 : 남자 = 여왕 : ? (정답은 여자)와 같은 단어 의미의 **유추작업**(Analogy task)에는 성능이 떨어짐

Word2Vec

: 실제값과 예측값에 대한 오차를 손실 함수를 통해 줄여나가며 학습하는 예측 기반의 방법론

장점: 예측 기반으로 단어 간 **유추 작업**에는 LSA보다 뛰어남

단점: 임베딩 벡터가 윈도우 크기 내에서만 주변 단어를 고려하기 때문에 코퍼스의 **전체적인 통계 정보**를 반영하지 못함

GloVe는 이러한 기존 방법론들의 각각의 한계를 지적하며, LSA의 메커니즘이었던 **카운트 기반의 방법**과 Word2Vec의 메커니즘이었던 **예측 기반의 방법론** 두 가지를 모두 사용함

2) 윈도우 기반 동시 등장 행렬 (Window based Co-occurrence Matrix)

: 단어의 **동시 등장 행렬**은 **행과 열을 전체 단어 집합의 단어들로 구성**하고,
i 단어의 윈도우 크기(Window Size) 내에서 **k 단어가 등장한 횟수**를 i행 k열에 기재한 행렬

> **예제**
> 아래와 같은 3개 문서로 구성된 텍스트 데이터가 있다고 해보자.

- I like deep learning
- I like NLP
- I enjoy flying

윈도우 크기가 N일 때는 **좌, 우에 존재하는 N개의 단어만 참고**하게 됨

윈도우 크기가 1일 때, 위의 텍스트를 가지고 구성한 동시 등장 행렬은 다음과 같음

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 행렬은 행렬을 전치(Transpose)해도 동일한 행렬이 된다는 특징이 있음

⇒ 그 이유는 i 단어의 윈도우 크기 내에서 k 단어가 등장한 빈도는
반대로 k 단어의 윈도우 크기 내에서 i 단어가 등장한 빈도와 동일하기 때문에!

3) 동시 등장 확률(Co-occurrence Probability)

동시 등장 확률 $P(k|i)$

: 동시 등장 행렬로부터 **특정 단어 i의 전체 등장 횟수를 카운트**하고,
특정 단어 i가 등장했을 때 어떤 단어 k가 등장한 횟수를 카운트하여 계산한 조건부 확률

$P(k|i)$ 에서 i를 중심 단어(Center Word), k를 주변 단어(Context Word)라고 했을 때,
위에서 배운 동시 등장 행렬에서 **중심 단어 i의 행의 모든 값을 더한 값을 분모로 하고 i행 k열의 값을 분자로 한 값**

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

GloVe의 제안 논문에서 가져온 동시 등장 확률을 표로 정리한 하나의 예

k가 solid일 때?!

$$\mathrm{P}(\mathrm{solid} \mid \mathrm{ice}) \approx \mathrm{P}(\mathrm{solid} \mid \mathrm{steam}) \times 8.9$$

solid는 '단단한'이라는 의미를 가졌으니까 '증기'라는 의미를 가지는 steam보다는 당연히 '얼음'이라는 의미를 가지는 ice라는 단어와 더 자주 등장할 것임

$$\mathrm{P}(\mathrm{solid} \mid \mathrm{ice}) / \mathrm{P}(\mathrm{solid} \mid \mathrm{steam}) \approx 8.9 > 1$$

그런데 k를 solid가 아니라 gas로 바꾸면?!

gas는 ice보다는 steam과 더 자주 등장하므로,

$$\mathrm{P}(\mathrm{gas} \mid \mathrm{ice}) / \mathrm{P}(\mathrm{gas} \mid \mathrm{steam}) \approx 0.085 < 1$$

동시 등장 확률과 크기 관계 비(ratio)	k=solid	k=gas	k=water	k=fasion
P(k | ice)	큰 값	작은 값	큰 값	작은 값
P(k | steam)	작은 값	큰 값	큰 값	작은 값
P(k | ice) / P(k | steam)	큰 값	작은 값	1에 가까움	1에 가까움

보기 쉽도록 조금 단순화해서 표현한 표

4) 손실 함수(Loss function)

▼ 용어 정리

- 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 I 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 I ice) / P(solid I steam) = 8.9
- w_i : 중심 단어 i의 임베딩 벡터
- $\tilde{w_k}$: 주변 단어 k의 임베딩 벡터

💡 GloVe의 아이디어를 한 줄로 요약하면,
'임베딩 된 중심 단어와 주변 단어 벡터의 내적이 전체 코퍼스에서의 동시 등장 확률이 되도록 만드는 것'
즉, 이를 만족하도록 **임베딩 벡터를 만드는 것이 목표!!**

손실함수는 다음과 같이 일반화 될 수 있음

$$\text{Loss function} = \sum_{m,n=1}^{V} (w_m^T \tilde{w_n} + b_m + \tilde{b_n} - logX_{mn})^2$$

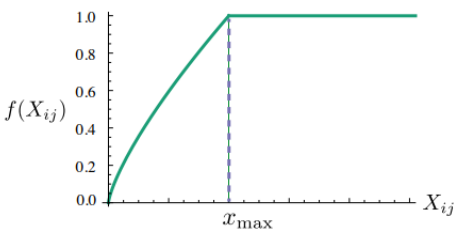
- $\tilde{b_k}$: $\tilde{w_k}$에 대한 편향
- V : 단어 집합의 크기

But, 아직 최적의 손실함수라고 하기에는 부족!

- GloVe 연구진은 $logX_{ik}$에서 X_{ik} 값이 0이 될 수 있음을 지적
  - 대안 : $logX_{ik}$를 $log(X_{ik}+1)$로 변경

But, 여전히 동시 등장 행렬 X는 **희소 행렬일 가능성**이 큼

- 동시 등장 행렬에는 많은 값이 0이거나, 동시 등장 빈도가 적어서 많은 값이 작은 수치를 가짐
  - 대안 : **X_{ik}에 영향 받는 가중치 함수 $f(X_{ik})$를 손실 함수에 도입**



X_{ik}의 값이 작으면 상대적으로 함수의 값은 작도록 하고, 값이 크면 함수의 값은 상대적으로 크도록!
하지만 X_{ik}가 지나치게 높다고해서 지나친 가중치를 주지 않기 위해서 또한 함수의 최대값 1이 정해져 있음
예를 들어 'It is'와 같은 불용어의 동시 등장 빈도수가 높다고해서 지나친 가중을 받아서는 안됨

이 함수의 값을 손실함수에 곱해주면 가중치의 역할을 할 수 있음

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

최종적으로 일반화 된 손실함수

$$\text{Loss function} = \sum_{m,n=1}^{V} f(X_{mn})(w_m^T \tilde{w_n} + b_m + \tilde{bn} - logX_{mn})^2$$

5) GloVe 훈련시키기

GloVe의 입력이 되는 훈련 데이터는 '영어와 한국어 Word2Vec 학습하기' 챕터에서 사용한 영어 데이터를 재사용. 모든 동일한 전처리를 마치고 이전과 동일하게 result에 결과가 저장되어있다고 가정.

```
# GloVe 패키지 설치
! pip install glove_python_binary

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 thread의 개수는 4로 설정, 에포크는 20
glove.fit(corpus.matrix, epochs=20, no_threads=4, verbose=True)
glove.add_dictionary(corpus.dictionary)

# 학습 완료
```

`glove.most_similar()` 는 입력 단어의 가장 유사한 단어들의 리스트를 리턴

```
print(glove.most_similar("man"))
# [('woman', 0.9621753707315267), ('guy', 0.8860281455579162), ('girl', 0.8609057388487154), ('kid', 0.8383640509911114)]

print(glove.most_similar("boy"))
# [('girl', 0.9436601252235809), ('kid', 0.8400949618225224), ('woman', 0.8397250531245034), ('man', 0.8303093585541573)]

print(glove.most_similar("water"))
# [('air', 0.838286550826724), ('clean', 0.8326093688298345), ('fresh', 0.8232884971285377), ('electricity', 0.8097066570385377)]
```

03 패스트 텍스트(FastText)

지금까지 배운 단어를 벡터로 만드는 방법 : 원-핫 인코딩, Word2Vec, GloVe
+ 페이스북에서 개발한 **FastText**

Word2Vec와 FastText와의 가장 큰 차이점

Word2Vec는 단어를 쪼개질 수 없는 단위로 생각

FastText는 하나의 단어 안에도 여러 단어들이 존재하는 것으로 간주
즉, **내부 단어(subword)**를 고려하여 학습

1) 내부 단어(subword) 학습

- 각 단어는 **글자 단위 n-gram의 구성**으로 취급
- n을 몇으로 결정하는지에 따라서 단어들이 얼마나 분리되는지 결정됨

ex) 예를 들어서 n을 3으로 잡은 트라이그램(tri-gram)의 경우,
apple은 app, ppl, ple로 분리하고 이들을 벡터로 ! → 5개 내부 단어(subword) 토큰을 벡터로!
+ 그리고 여기에 추가적으로 하나를 더 벡터화!

```
# n = 3인 경우
<ap, app, ppl, ple, le>, <apple>
```

여기서 “내부 단어들을 벡터화한다”는 의미는 저 단어들에 대해서 Word2Vec을 수행한다는 의미
위와 같이 내부 단어들의 벡터값을 얻었다면, 단어 apple의 벡터값은 저 위 벡터값들의 총 합으로 구성됨

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

그리고 이런 방법은 Word2Vec에서는 얻을 수 없었던 강점을 가짐

2) 모르는 단어(Out Of Vocabulary, OOV)에 대한 대응

- FastText의 인공 신경망을 학습한 후에는 데이터 셋의 모든 단어의 각 n-gram에 대해서 워드 임베딩 됨
- 데이터 셋만 충분한다면 위와 같은 내부 단어(Subword)를 통해 모르는 단어(Out Of Vocabulary, OOV)에 대해서도 다른 단어와의 유사도를 계산할 수 있다는 장점!

> 예시) FastText에서 birthplace(출생지)란 단어를 학습하지 않은 상태라고 가정해보자.

하지만 다른 단어에서 birth와 place라는 내부 단어가 있었다면, FastText는 birthplace의 벡터를 얻을 수 있음!
⇒ 이는 모르는 단어에 제대로 대처할 수 없는 Word2Vec, GloVe와는 다른 점

3) 단어 집합 내 빈도 수가 적었던 단어(Rare Word)에 대한 대응

- Word2Vec의 경우에는 등장 빈도 수가 적은 단어(rare word)에 대해서는 임베딩의 정확도가 높지 않음
  ← 참고할 수 있는 경우의 수가 적다보니 정확하게 임베딩이 되지 않는 경우
- 하지만 FastText의 경우, 만약 단어가 희귀 단어라도, 그 단어의 n-gram이 다른 단어의 n-gram과 겹치는 경우라면, Word2Vec과 비교하여 비교적 높은 임베딩 벡터값을 얻음!
  ⇒ FastText가 노이즈가 많은 코퍼스에서 강점을 가진 것 또한 이와 같은 이유
- 실제 많은 비정형 데이터에는 오타가 섞여있고 오타가 섞인 단어는 당연히 일종의 희귀 단어가 됨

  ⇒ 즉, Word2Vec에서는 오타가 섞인 단어는 임베딩이 제대로 되지 않지만,
  FastText는 이에 대해서도 일정 수준의 성능을 보임

  → 예를 들어 단어 apple과 오타로 p를 한 번 더 입력한 appple의 경우에는
  실제로 많은 개수의 동일한 n-gram을 가질 것임

4) 실습으로 비교하는 Word2Vec Vs. FastText

(1) Word2Vec

(이전 Word2Vec의 실습의 전처리 코드와 Word2Vec 학습 코드를 그대로 수행했음을 가정)

입력 단어에 대해서 유사한 단어를 찾아내는 코드에 이번에는 electrofishing이라는 단어를 넣어보면

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

해당 코드는 정상 작동하지 않고 에러 발생

```
KeyError: "word 'electrofishing' not in vocabulary"
# 단어 집합(Vocabulary)에 electrofishing이 존재하지 않음
```

이처럼 Word2Vec는 학습 데이터에 존재하지 않는 단어 즉, 모르는 단어에 대해서는 임베딩 벡터가 존재하지 않기 때문에 단어의 유사도를 계산할 수 없음

(2) FastText

이번에는 전처리 코드는 그대로 사용하고 Word2Vec 학습 코드만 FastText 학습 코드로 변경하여 실행

```
from gensim.models import FastText

model = FastText(result, size=100, window=5, min_count=5, workers=4, sg=1)
```

```
# electrofishing에 대해서 유사 단어를 찾기
model.wv.most_similar("electrofishing")
```

```
[('electrolux', 0.7934642434120178), ('electrolyte', 0.78279709815979), ('electro', 0.779127836227417), ('electric', 0.7753111
720085144), ('airbus', 0.7648627758026123), ('fukushima', 0.7612422704696655), ('electrochemical', 0.7611693143844604), ('gast
ric', 0.7483425140380859), ('electroshock', 0.7477173805236816), ('overfishing', 0.7435552477836609)]
```

Word2Vec는 학습하지 않은 단어에 대해서 유사한 단어를 찾아내지 못 했지만, FastText는 유사한 단어를 계산해서 출력함

5) 한국어에서의 FastText

한국어의 경우에도 OOV 문제를 해결하기 위해 FastText를 적용하고자 함

(1) 음절 단위

예를 들어서 음절 단위의 임베딩의 경우에 n=3일때 '자연어처리'라는 단어에 대해 n-gram을 만들어보면

```
<자연, 자연어, 연어처, 어처리, 처리>
```

(2) 자모 단위(초성, 중성, 종성 단위)

이제 더 나아가 자모 단위(초성, 중성, 종성 단위)로 임베딩

- 음절 단위가 아니라, 자모 단위로 가게 되면 오타나 노이즈 측면에서 더 강한 임베딩을 기대함

예를 들어 '자연어처리'라는 단어에 대해서 초성, 중성, 종성을 분리하고, 만약, 종성이 존재하지 않는다면
'_'라는 토큰을 사용한다고 가정한다면 '자연어처리'라는 단어는 아래와 같이 분리가 가능

```
분리된 결과 : ㅈ ㅏ _ ㅇ ㅕ ㄴ ㅇ ㅓ _ ㅊ ㅓ _ ㄹ ㅣ _
```

그리고 분리된 결과에 대해서 n=3일 때, n-gram을 적용하여, 임베딩을 한다면 다음과 같음

```
< ㅈ ㅏ, ㅈ ㅏ _, ㅏ _ ㅇ, ... 중략>
```

<자모 단위 한국어 FastText 학습하기>는 페이지 삭제됨

04 사전 훈련된 워드 임베딩(Pre-trained Word Embedding)

케라스의 임베딩 층(embedding layer) 과 **사전 훈련된 워드 임베딩(pre-trained word embedding)**

- **훈련 데이터의 단어들을** 임베딩 층(embedding layer)을 구현하여 **임베딩 벡터로 학습**
  ⇒ keras의 `Embedding()` 사용
- 위키피디아 등과 같은 방대한 코퍼스를 가지고 Word2vec, FastText, GloVe 등을 통해서 **미리 훈련된 임베딩 벡터를 불러오는 방법**을 사용하는 경우도 있음

1) 케라스 임베딩 층(Keras Embedding layer)

`Embedding()`은 인공 신경망 구조 관점에서 임베딩 층(embedding layer) 구현

(1) 임베딩 층은 룩업 테이블이다.

임베딩 층의 입력으로 사용하기 위해서 입력 시퀀스의 각 단어들은 모두 정수 인코딩이 되어있어야 함

- 어떤 단어 → 단어에 부여된 고유한 정수값 → 임베딩 층 통과 → 밀집 벡터

임베딩 층은 입력 정수에 대해 밀집 벡터(dense vector)로 맵핑

→ 이 **밀집 벡터**는 인공 신경망의 학습 과정에서 가중치가 학습되는 것과 같은 방식으로 훈련됨

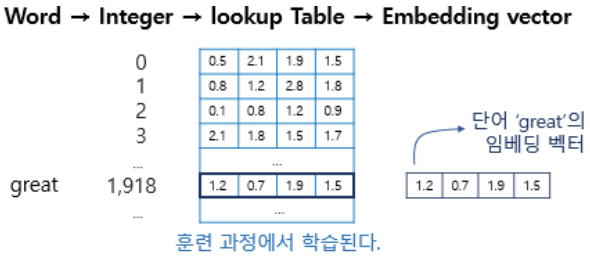
→ 훈련 과정에서 단어는 모델이 풀고자하는 작업에 맞는 값으로 **업데이트 됨**

→ 이 밀집 벡터를 **임베딩 벡터**라고 부름

정수를 밀집 벡터 또는 임베딩 벡터로 맵핑한다는 것은 어떤 의미일까?

특정 단어와 맵핑되는 정수를 인덱스로 가지는 테이블로부터 임베딩 벡터 값을 가져오는 **룩업 테이블**임

그리고 이 테이블은 단어 집합의 크기만큼의 행을 가지므로 **모든 단어는 고유한 임베딩 벡터를 가짐**



- 위의 그림에서는 임베딩 벡터의 차원이 4로 설정되어져 있음
- 단어 great은 정수 인코딩 과정에서 1,918의 정수로 인코딩이 되었고 그에 따라 단어 집합의 크기만큼의 행을 가지는 테이블에서 인덱스 1,918번에 위치한 행을 단어 great의 임베딩 벡터로 사용
- 이 임베딩 벡터는 모델의 입력이 되고, 역전파 과정에서 단어 great의 임베딩 벡터값이 학습됨

💡 케라스는
❌ : 단어를 정수 인덱스로 바꾸고 → 원-핫 벡터로 변환 후 → 임베딩 층의 입력으로 사용
⭕ : 단어를 정수 인코딩까지만 진행 후 → 임베딩 층의 입력으로 사용하여 룩업 테이블 결과인 임베딩 벡터를 리턴

[케라스의 임베딩 층 구현 코드]

```
vocab_size = 20000
output_dim = 128
input_length = 500

v = Embedding(vocab_size, output_dim, input_length=input_length)
```

- 임베딩 층의 세 개의 인자

  `vocab_size` = 텍스트 데이터의 전체 단어 집합의 크기

  `output_dim` = 워드 임베딩 후의 임베딩 벡터의 차원

  `input_length` = 입력 시퀀스의 길이

(2) 임베딩 층 사용하기

[문장의 긍, 부정을 판단하는 감성 분류 모델] (긍정: 1, 부정: 0)

```
import numpyas np
from tensorflow.keras.preprocessing.textimport Tokenizer
from tensorflow.keras.preprocessing.sequenceimport pad_sequences

sentences = ['nice great best amazing', 'stop lies', 'pitiful nerd', 'excellent work', 'supreme quality', 'bad', 'highly respe
ctable']
y_train = [1, 0, 0, 1, 1, 0, 1]
```

```
# 케라스의 토크나이저 사용하여 단어 집합 만들고, 크기 확인
tokenizer = Tokenizer()
tokenizer.fit_on_texts(sentences)
vocab_size = len(tokenizer.word_index) + 1 # 패딩을 고려하여 +1
print('단어 집합 :',vocab_size)
```

```
단어 집합 : 16
```

```
# 각 문장에 대해서 정수 인코딩
X_encoded = tokenizer.texts_to_sequences(sentences)
print('정수 인코딩 결과 :',X_encoded)
```

```
정수 인코딩 결과 : [[1, 2, 3, 4], [5, 6], [7, 8], [9, 10], [11, 12], [13], [14, 15]]
```

```
# 가장 길이가 긴 문장의 길이 구하기
max_len = max(len(l) for l in X_encoded)
print('최대 길이 :',max_len)
```

```
최대 길이 : 4
```

```
# 최대 길이로 모든 샘플에 대해서 패딩 진행
X_train = pad_sequences(X_encoded, maxlen=max_len, padding='post')
y_train = np.array(y_train)
print('패딩 결과 :')
print(X_train)
```

```
패딩 결과 :
[[ 1  2  3  4]
 [ 5  6  0  0]
 [ 7  8  0  0]
 [ 9 10  0  0]
 [11 12  0  0]
 [13  0  0  0]
 [14 15  0  0]]
# 전처리 끝
```

```
# 전형적인 이진 분류 모델 설계
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Embedding, Flatten

embedding_dim = 4

model = Sequential()
model.add(Embedding(vocab_size, embedding_dim, input_length=max_len))
model.add(Flatten())
model.add(Dense(1, activation='sigmoid')) # 활성화 함수로 시그모이드 함수

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

```
# 손실함수로 binary_crossentropy
model.fit(X_train, y_train, epochs=100, verbose=2)
```

학습 과정에서 현재 각 단어들의 임베딩 벡터들의 값은 출력층의 가중치와 함께 학습됨

2) 사전 훈련된 워드 임베딩(Pre-Trained Word Embedding) 사용하기

보다 많은 훈련 데이터로 이미 Word2Vec이나 GloVe 등으로 **학습되어져 있는 임베딩 벡터들을 사용**
⇒ 성능의 개선을 가져올 수 있음

[사전 훈련된 GloVe와 Word2Vec 임베딩을 사용해서 모델을 훈련시키는 실습]

훈련 데이터는 앞서 사용했던 데이터에 동일한 전처리까지 진행된 상태라고 가정

```
print(X_train)
```

```
[[ 1  2  3  4]
 [ 5  6  0  0]
 [ 7  8  0  0]
 [ 9 10  0  0]
 [11 12  0  0]
 [13  0  0  0]
 [14 15  0  0]]
```

```
print(y_train)
```

```
[1, 0, 0, 1, 1, 0, 1]
```

(1) 사전 훈련된 GloVe 사용하기

```
# glove에 있는 모든 임베딩 벡터들을 불러와 로드한 임베딩 벡터의 개수를 확인
embedding_dict = dict()

f =open('glove.6B.100d.txt', encoding="utf8")

for line in f:
    word_vector = line.split()
    word = word_vector[0]

    # 100개의 값을 가지는 array로 변환
    word_vector_arr = np.asarray(word_vector[1:], dtype='float32')
    embedding_dict[word] = word_vector_arr
f.close()

print('%s개의 Embedding vector가 있습니다.' % len(embedding_dict))
```

```
400000개의 Embeddingvector가 있습니다.
```

```
# 임의의 단어 'respectable'의 임베딩 벡터값과 크기를 출력
print(embedding_dict['respectable'])
print('벡터의 차원 수 :',len(embedding_dict['respectable']))
```

```
[-0.049773   0.19903    0.10585 ... 중략 ... -0.032502   0.38025  ]
벡터의 차원 수 : 100
```

풀고자 하는 문제의 단어 집합 크기의 행과 100개의 열을 가지는 행렬 생성

이 행렬의 값은 전부 0으로 채우고, 이 행렬에 사전 훈련된 임베딩 값을 넣어줄 것임

```
embedding_matrix = np.zeros((vocab_size, 100))
print('임베딩 행렬의 크기(shape) :',np.shape(embedding_matrix)
```

```
임베딩 행렬의 크기(shape) : (16, 100)
```

```
# 기존 데이터의 각 단어와 맵핑된 정수값 확인
print(tokenizer.word_index.items())
```

```
dict_items([('nice', 1), ('great', 2), ('best', 3), ('amazing', 4), ('stop', 5), ('lies', 6), ('pitiful', 7), ('nerd', 8), ('e
xcellent', 9), ('work', 10), ('supreme', 11), ('quality', 12), ('bad', 13), ('highly', 14), ('respectable', 15)])
```

```
# 사전 훈련된 GloVe에서 'great'의 벡터값을 확인
print(embedding_dict['great'])
```

```
[-0.013786   0.38216    0.53236    0.15261   -0.29694   -0.20558
.. 중략 ...
 -0.69183   -1.0426     0.28855    0.63056  ]
```

단어 집합의 모든 단어에 대해서 사전 훈련된 GloVe의 임베딩 벡터들을 맵핑한 후 'great'의 벡터값이 의도한 인덱스의 위치에 삽입되었는지 확인

```
for word, indexin tokenizer.word_index.items():
    # 단어와 맵핑되는 사전 훈련된 임베딩 벡터값
    vector_value = embedding_dict.get(word)
if vector_valueisnotNone:
        embedding_matrix[index] = vector_value
```

embedding_matrix의 인덱스 2에서의 값을 확인

```
embedding_matrix[2]
```

```
array([-0.013786  ,  0.38216001,  0.53236002,  0.15261   , -0.29694   ,
        ... 중략 ...
       -0.39346001, -0.69182998, -1.04260004,  0.28854999,  0.63055998])
```

이전에 확인한 사전에 훈련된 GloVe에서의 'great'의 벡터값과 일치

(2) 사전 훈련된 Word2Vec 사용하기

구글의 사전 훈련된 Word2Vec 모델을 로드하여 word2vec_model에 저장 후 크기를 확인

```
모델의 크기(shape) : (3000000, 300)
```

300의 차원을 가진 Word2Vec 벡터가 3,000,000개

모든 값이 0으로 채워진 임베딩 행렬을 만들어줌

풀고자 하는 문제의 단어 집합 크기의 행과 300개의 열을 가지는 행렬 생성

이 행렬의 값은 전부 0으로 채우고 이 행렬에 사전 훈련된 임베딩 값을 넣어줄 것임

```
embedding_matrix = np.zeros((vocab_size, 300))
print('임베딩 행렬의 크기(shape) :',np.shape(embedding_matrix)
```

```
임베딩 행렬의 크기(shape) : (16, 300)
```

word2vec_model에서 특정 단어를 입력하면 해당 단어의 임베딩 벡터를 리턴받을텐데,
만약 word2vec_model에 특정 단어의 임베딩 벡터가 없다면 None을 리턴하도록 하는 함수 `get_vector()`를 구현

```
def get_vector(word):
  if wordin word2vec_model:
    return word2vec_model[word]
  else:
  return None
```

단어 집합으로부터 단어를 1개씩 호출하여 word2vec_model에 해당 단어의 임베딩 벡터값이 존재하는지 확인

만약 None이 아니라면 존재한다는 의미이므로, 임베딩 행렬에 해당 단어의 인덱스 위치의 행에 임베딩 벡터의 값을 저장

```
for word, index in tokenizer.word_index.items():
    # 단어와 맵핑되는 사전 훈련된 임베딩 벡터값
    vector_value = get_vector(word)
if vector_value is not None:
        embedding_matrix[index] = vector_value
```

16개의 단어와 맵핑되는 임베딩 행렬이 완성됨

제대로 맵핑이 됐는지 확인

```
# 기존에 word2vec_model에 저장되어 있던 단어 'nice'의 임베딩 벡터값을 확인
print(word2vec_model['nice'])
```

```
[ 0.15820312  0.10595703 -0.18945312  0.38671875  0.08349609 -0.26757812
  0.08349609  0.11328125 -0.10400391  0.17871094 -0.12353516 -0.22265625
  ... 중략 ...
 -0.16894531 -0.08642578 -0.08544922  0.18945312 -0.14648438  0.13476562
 -0.04077148  0.03271484  0.08935547 -0.26757812  0.00836182 -0.21386719]
```

```
# 단어 'nice'의 맵핑된 정수를 확인
print('단어 nice의 맵핑된 정수 :', tokenizer.word_index['nice'])
```

```
단어 nice의 맵핑된 정수 : 1
```

1의 값을 가지므로 embedding_matirx의 1번 인덱스에는 단어 'nice'의 임베딩 벡터값이 있어야 함

```
print(embedding_matrix[1])
```

```
[ 0.15820312  0.10595703 -0.18945312  0.38671875  0.08349609 -0.26757812
  0.08349609  0.11328125 -0.10400391  0.17871094 -0.12353516 -0.22265625
  ... 중략 ...
 -0.16894531 -0.08642578 -0.08544922  0.18945312 -0.14648438  0.13476562
 -0.04077148  0.03271484  0.08935547 -0.26757812  0.00836182 -0.21386719]
```

05 엘모(Embeddings from Language Model, ELMo)

ELMo(Embeddings from Language Model)는 2018년에 제안된 새로운 워드 임베딩 방법론으로, 해석하면 '언어 모델로 하는 임베딩'

ELMo의 가장 큰 특징은 **사전 훈련된 언어 모델(Pre-trained language model)**을 사용한다는 점!

1) ELMo(Embeddings from Language Model)

> Bank라는 단어를 생각해보자.

Bank Account(은행 계좌)와 River Bank(강둑)에서의 Bank는 전혀 다른 의미를 가지는데, Word2Vec이나 GloVe 등으로 표현된 임베딩 벡터들은 이를 제대로 반영하지 못한다는 단점이 있음

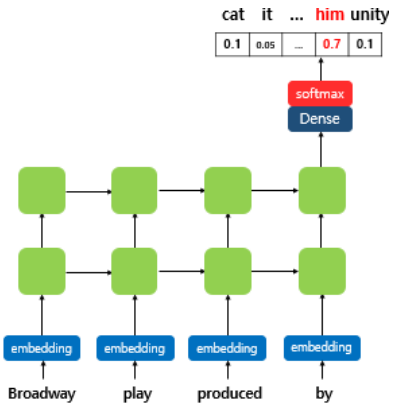
예를 들어서 Word2Vec이나 GloVe 등의 임베딩 방법론으로 Bank란 단어를 [0.2 0.8 -1.2]라는 임베딩 벡터로 임베딩하였다고 하면, 이 단어는 Bank Account(은행 계좌)와 River Bank(강둑)에서의 Bank는 전혀 다른 의미임에도 불구하고 두 가지 상황 모두에서 [0.2 0.8 -1.2]의 벡터가 사용될 것임..

같은 표기의 단어라도 문맥에 따라서 다르게 워드 임베딩을 할 수 있으면 자연어 처리의 성능을 올릴 수 있을 것임!

⇒ 워드 임베딩 시 문맥을 고려해서 임베딩을 하겠다는 아이디어가 **문맥을 반영한 워드 임베딩(Contextualized Word Embedding)**

2) biLM(Bidirectional Language Model)의 사전 훈련

[다음 단어를 예측하는 작업인 언어 모델링]

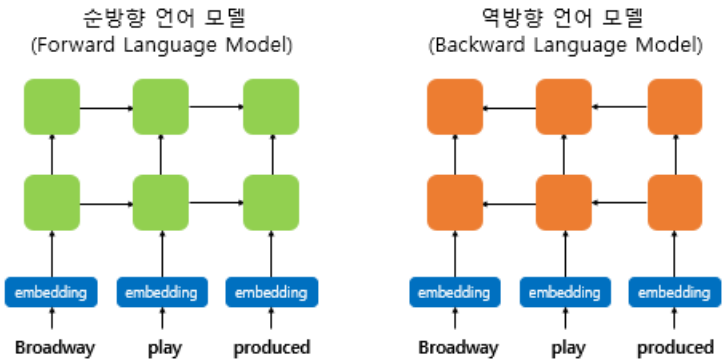


은닉층이 2개인 일반적인 단방향 RNN 언어 모델의 언어 모델링

- RNN 언어 모델은 문장으로부터 단어 단위로 입력을 받는데, RNN 내부의 은닉 상태 h_t는 시점(time step)이 지날수록 점점 업데이트 됨
- 이는 결과적으로 RNN의 h_t의 값이 문장의 문맥 정보를 점차적으로 반영

- 그런데 ELMo는 위의 그림의 순방향 RNN 뿐만 아니라, 위의 그림과는 **반대 방향으로 문장을 스캔하는 역방향 RNN 또한 활용**
- ELMo는 양쪽 방향의 언어 모델을 둘 다 학습하여 활용해서 이 언어 모델을 **biLM(Bidirectional Language Model)** 이라고 함

ELMo에서 말하는 biLM은 기본적으로 다층 구조(Multi-layer)를 전제 (은닉층이 최소 2개 이상이라는 의미!)



은닉층이 2개인 순방향 언어 모델과 역방향 언어 모델

이때 **biLM의 각 시점의 입력이 되는 단어 벡터**는

❌

임베딩 층(embedding layer)을 사용해서 얻은 것

⭕

합성곱 신경망을 이용한 문자 임베딩(character embedding)을 통해 얻은 단어 벡터

문자 임베딩은 마치 서브단어(subword)의 정보를 참고하는 것처럼, 문맥과 상관없이 dog란 단어와 doggy란 단어의 연관성을 찾아낼 수 있음. 또한 이 방법은 OOV에도 견고한다는 장점이 있음
여기서는 임베딩층, Word2Vec 등 외에 단어 벡터를 얻는 또 다른 방식도 있다고만 알아두자!

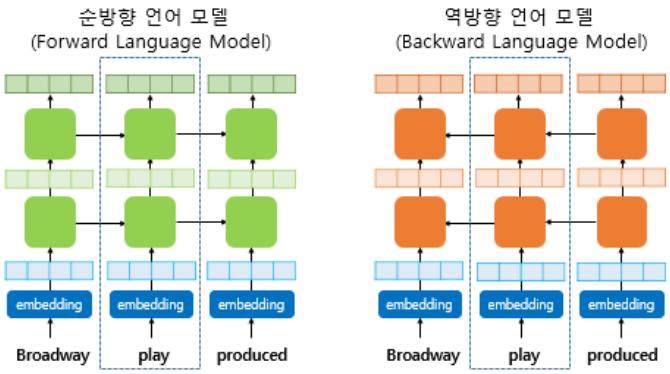
💡 주의할 점은 앞서 설명한 **양방향 RNN**과 **ELMo에서의 biLM**은 다르다는 것!

양방향 RNN은 순방향 RNN의 **은닉 상태와** 역방향의 RNN의 **은닉 상태를 연결(concatenate)하여 다음층의 입력으로 사용함**

반면, biLM의 순방향 언어모델과 역방향 언어모델이라는 두 개의 언어 모델을 **별개의 모델로 보고 학습함**

3) biLM의 활용

[biLM이 언어 모델링을 통해 학습된 후
ELMo가 사전 훈련된 biLM을 통해 입력 문장으로부터 단어를 임베딩하기 위한 과정]



> 예제) play란 단어가 임베딩이 되고 있다는 가정 하에 ELMo를 설명해보자.

play라는 단어를 임베딩 하기 위해서 ELMo는 위의 점선의 사각형 내부의 **각 층의 결과값**을 재료로 사용

다시 말해 **해당 시점(time step)의 BiLM의 각 층의 출력값을 가져옴**

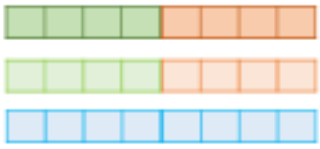
그리고 순방향 언어 모델과 역방향 언어 모델의 **각 층의 출력값**을 연결(concatenate)하고 추가 작업을 진행

여기서 **각 층의 출력값**이란 첫번째는 임베딩 층을 말하며, 나머지 층은 각 층의 은닉 상태를 말함

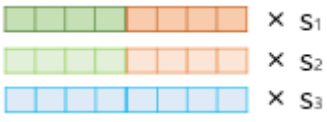
ELMo의 직관적인 아이디어
: **각 층의 출력값이 가진 정보는 전부 서로 다른 종류의 정보를 갖고 있을 것이므로, 이들을 모두 활용한다!!**

[ELMo가 임베딩 벡터를 얻는 과정]

(1) 각 층의 출력값을 연결(concatenate)하기

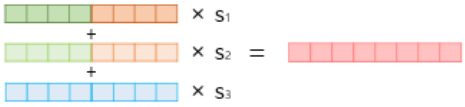


(2) 각 층의 출력값 별로 가중치 주기



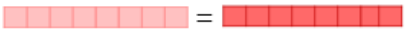
이 가중치를 여기서는 s1, s2, s3라고 하자

(3) 각 층의 출력값을 모두 더하기



cf) 2)번과 3)번의 단계를 요약하여 가중합(Weighted Sum)을 한다고 함

(4) 벡터의 크기를 결정하는 스칼라 매개변수(λ)를 곱하기

$\gamma \times$  $=$

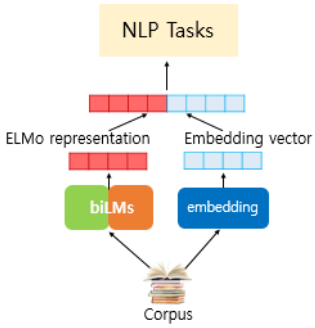
이렇게 완성된 벡터: ELMo 표현(representation)

⇒ 지금까지는 **ELMo 표현**을 얻기 위한 과정

[ELMo를 입력으로 사용하고 수행하고 싶은 텍스트 분류, 질의 응답 시스템 등의 자연어 처리 작업]

> 예시) 텍스트 분류 작업을 하고 싶다고 가정하자.
> ELMo 표현을 어떻게 텍스트 분류 작업에 사용할 수 있을까?

- 텍스트 분류 작업을 위해서 GloVe와 같은 기존의 방법론을 사용한 임베딩 벡터 준비
- 이때, GloVe를 사용한 임베딩 벡터만 텍스트 분류 작업에 사용하는 것이 아니라, 이렇게 준비된 **ELMo 표현을 GloVe 임베딩 벡터와 연결(concatenate)해서 입력으로 사용**할 수 있음



ELMo 표현이 기존의 GloVe 등과 같은 임베딩 벡터와 함께 NLP 태스크의 입력이 됨

4) ELMo 표현을 사용해서 스팸 메일 분류하기

```
import tensorflow_hubas hub
import tensorflowas tf
from kerasimport backendas K
import urllib.request
import pandasas pd
import numpyas np
```

텐서플로우 허브로부터 ELMo를 다운로드

```
elmo = hub.Module("https://tfhub.dev/google/elmo/1", trainable=True)
# 텐서플로우 허브로부터 ELMo를 다운로드

sess = tf.Session()
K.set_session(sess)
sess.run(tf.global_variables_initializer())
sess.run(tf.tables_initializer())
```

데이터를 불러오고, 5개만 출력

```
# 스팸 메일 분류하기 데이터를 다운로드
urllib.request.urlretrieve("https://raw.githubusercontent.com/mohitgupta-omg/Kaggle-SMS-Spam-Collection-Dataset-/master/spam.csv", filename="spam.csv")
data = pd.read_csv('spam.csv', encoding='latin-1')
data[:5]
```

	v1	v2	Unnamed: 2	Unnamed: 3	Unnamed: 4
0	ham	Go until jurong point, crazy.. Available only ...	NaN	NaN	NaN
1	ham	Ok lar... Joking wif u oni...	NaN	NaN	NaN
2	spam	Free entry in 2 a wkly comp to win FA Cup fina...	NaN	NaN	NaN
3	ham	U dun say so early hor... U c already then say...	NaN	NaN	NaN
4	ham	Nah I don't think he goes to usf, he lives aro...	NaN	NaN	NaN

cf) spam = Junk email = Bad email ⇒ 1 ham = Not spam = Good email ⇒ 0

```
# 필요한 건 v2열과 v1열
data['v1'] = data['v1'].replace(['ham','spam'],[0,1]) # 숫자 레이블로 변경
y_data =list(data['v1'])
X_data =list(data['v2'])
```

```
print(len(X_data))
n_of_train =int(len(X_data) * 0.8) # 훈련 데이터와 테스트 데이터를 8:2 비율로 분할
n_of_test =int(len(X_data) - n_of_train)
print(n_of_train)
print(n_of_test) # 전체 데이터 개수의 80%와 20%는 각각 몇 개인지 확인
```

```
5572
4457
1115
```

```
X_train = np.asarray(X_data[:n_of_train]) #X_data 데이터 중에서 앞의 4457개의 데이터만 저장
y_train = np.asarray(y_data[:n_of_train]) #y_data 데이터 중에서 앞의 4457개의 데이터만 저장
X_test = np.asarray(X_data[n_of_train:]) #X_data 데이터 중에서 뒤의 1115개의 데이터만 저장
y_test = np.asarray(y_data[n_of_train:]) #y_data 데이터 중에서 뒤의 1115개의 데이터만 저장
```

⇒ 훈련을 위한 데이터 준비는 끝!

ELMo와 설계한 모델을 연결하는 작업들 진행

⇒ ELMo는 텐서플로우 허브로부터 가져온 것이기 때문에,
케라스에서 사용하기 위해서는 케라스에서 사용할 수 있도록 변환해주는 작업들이 필요함

```
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
```

모델 설계

```
from keras.modelsimport Model
from keras.layersimport Dense, Lambda, Input

input_text = Input(shape=(1,), dtype=tf.string)
embedding_layer = Lambda(ELMoEmbedding, output_shape=(1024, ))(input_text)
# ELMo를 이용한 임베딩 층을 거쳐서
hidden_layer = Dense(256, activation='relu')(embedding_layer)
# 256개의 뉴런이 있는 은닉층을 거친 후
output_layer = Dense(1, activation='sigmoid')(hidden_layer)
# 마지막 1개의 뉴런을 통해 이진 분류를 수행/ 활성화 함수:시그모이드 함수
```

```
model = Model(inputs=[input_text], outputs=output_layer)
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy']) # 손실함수:binary_crossentropy
```

```
history = model.fit(X_train, y_train, epochs=1, batch_size=60)
```

```
Epoch 1/1
4457/4457 [==============================] - 1508s 338ms/step - loss: 0.1129 - acc: 0.9619
# 훈련 데이터에서는 정확도 96%
```

```
print("\n 테스트 정확도: %.4f" % (model.evaluate(X_test, y_test)[1]))
```

```
1115/1115 [==============================] - 381s 342ms/step
테스트 정확도: 0.9803
# 1번의 에포크에서 테스트 데이터 정확도 98%
```