위키독스 9.4 ~ 10.2

2017010715허지혜

01. **영어/한국어** Word2Vec **훈련**

02. **글로브**(GloVe)

03. **파이토치**(PyTorch)**의** nn.Embedding()

04. **순환신경망**(Recurrent Neural Network)

05. **장단기 메모리**(Long Short-Term Memory)

01. 영어/한국어 Word2Vec 훈련

1. 영어 Word2Vec 만들기

```
<file id="1">
  <head>
<url>http://www.ted.com/talks/knut_haanaes_two_reasons_companies_fail_and_how_to_avoid_them</url>
      <pagesize>72832</pagesize>
... xml 문법 중략 ...
<content>
Here are two reasons companies fail: they only do more of the same, or they only do what's new.
To me the real, real solution to quality growth is figuring out the balance between two activities:
... content 내용 중략 ...
To me, the irony about the Facit story is hearing about the Facit engineers, who had bought cheap, small electronic calculators in Japan that they used to double-check their calculators.
(Laughter)
... content 내용 중략 ...
(Applause)
</content>
</file>
<file id="2">
    <head>
<url>http://www.ted.com/talks/lisa_nip_how_humans_could_evolve_to_survive_in_space<url>
... 이하 중략 ...
```

Xml 파일

```
targetXML=open('ted_en-20160408.xml', 'r', encoding='UTF8')

target_text = etree.parse(targetXML)
parse_text = '₩n'.join(target_text.xpath('//content/text()'))
```

<content>와 </content> 사이 내용만 가져오기

```
content_text = re.sub(r'₩([^)]*₩)', '', parse_text)
```

괄호 구성 내용 제거

```
sent_text = sent_tokenize(content_text)
```

NLTK를 이용해 문장 토큰화 수행

```
normalized_text = []
for string in sent_text:
     tokens = re.sub(r"[^a-z0-9]+", " ", string.lower())
     normalized_text.append(tokens)
```

각 문장 구두점 제거, 대문자 소문자 변환

```
result = []
result = [word_tokenize(sentence) for sentence in normalized_text]
```

각 문장 NLTK를 이용해 단어 토큰화 수행

```
print('총 샘플의 개수 : {}'.format(len(result)))
```

```
총 샘플의 개수 : 273424
```

01. 영어/한국어 Word2Vec 훈련

1. 영어 Word2Vec 만들기

```
for line in result[:3]: # 샘플 3개만 출력
    print(line)
```

토큰화가 잘 수행되었다.

```
['here', 'are', 'two', 'reasons', 'companies', 'fail', 'they', 'only', 'do', 'mor
e', 'of', 'the', 'same', 'or', 'they', 'only', 'do', 'what', 's', 'new']
['to', 'me', 'the', 'real', 'real', 'solution', 'to', 'quality', 'growth', 'is', 'f
iguring', 'out', 'the', 'balance', 'between', 'two', 'activities', 'exploration',
'and', 'exploitation']
['both', 'are', 'necessary', 'but', 'it', 'can', 'be', 'too', 'much', 'of', 'a', 'g
ood', 'thing']
```

```
from gensim.models import Word2Vec, KeyedVectors
model = Word2Vec(sentences=result, size=100, window=5, min_count=5, workers=4,
sg=0)
```

size = 워드 벡터의 특징 값. 즉, 임베딩 된 벡터의 차원.
window = 윈도우 크기
min_count = 단어 최소 빈도 수 제한
workers = 학습을 위한 프로세스 수
sg = 0은 CBOW, 1은 Skip-gram.

```
model_result = model.wv.most_similar("man")
print(model_result)
```

[('woman', 0.842622697353363),
('guy', 0.8178728818893433),
('boy', 0.7774451375007629),
('lady', 0.7767927646636963),
('girl', 0.7583760023117065),
('gentleman',
0.7437191009521484),
('soldier', 0.7413754463195801),
('poet', 0.7060446739196777),
('kid', 0.6925194263458252),
('friend', 0.6572611331939697)]

01. 영어/한국어 Word2Vec 훈련

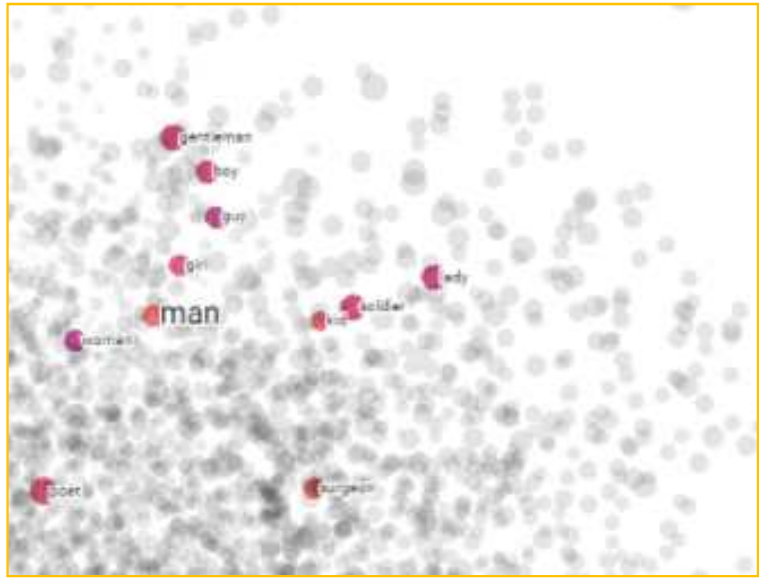
1. 영어 Word2Vec 만들기

```
model.wv.save_word2vec_format('./eng_w2v') # 모델 저장
loaded_model = KeyedVectors.load_word2vec_format("eng_w2v") # 모델 로드
```

```
model_result = loaded_model.most_similar("man")
print(model_result)
```

```
[('woman', 0.849030613899231), ('guy', 0.8245782852172852), ('boy', 0.7727351188659668), ('lady', 0.7713152766227722), ('girl', 0.7478859424591064), ('gentleman', 0.7357934713363647), ('soldier', 0.7194231152534485), ('kid', 0.7184772491455078), ('poet', 0.6919025182723999), ('friend', 0.6493204236030579)]
```

임베딩 프로젝터
https://projector.tensorflow.org/



01. 영어/한국어 Word2Vec 훈련

2. 사전 훈련된 Word2Vec 임베딩 영어

구글이 제공하는 사전 훈련된 Word2Vec 모델을 사용

⇒ 3백만 개의 Word2Vec 단어 벡터들을 제공하고
임베딩 벡터의 차원은 300이다.

사용 방법은 모델을 다운로드하고 파일 경로를 기재
https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit

```
import gensim

# 구글의 사전 훈련된 Word2Vec 모델을 로드합니다.
model = gensim.models.KeyedVectors.load_word2vec_format('GoogleNews-vectors-negative300.bin 파일 경로', binary=True)
```

```
print (model.similarity('this', 'is')) # 두 단어의 유사도 계산하기
print (model.similarity('post', 'book'))
```

```
0.407970363878
0.0572043891977
```

```
print(model['book']) # 단어 'book'의 벡터 출력
```

```
[ 0.11279297 -0.02612305 -0.04492188  0.06982422  0.140625    0.03039551
 -0.04370117  0.24511719  0.08740234 -0.05053711  0.23144531 -0.07470703
... 300개의 값이 출력되는 관계로 중략 ...
  0.03637695 -0.16796875 -0.01483154  0.09667969 -0.05761719 -0.00515747]
```

02. 글로브(GloVe)

글로브(Global Vectors for Word Representation)

카운트 기반과 예측 기반을 모두 사용하는 방법론으로 2014년 미국 스탠포드대학에서 개발한 단어 임베딩 방법론

기존의 카운트 기반과 예측 기반의 단점을 지적하며 보완한다는 목적으로 나왔으며 실제로 Word2Vec 만큼 뛰어난 성능을 보여줌

LSA

각 단어의 빈도수를 카운트한 행렬을 입력 받아 차원을 축소해 잠재된 의미를 이끔

Word2Vec

주변을 고려해 유추 작업이 뛰어남

임베딩된 두 단어 벡터의 내적이 말뭉치 전체에서의 동시등장확률 로그값이 되도록 목적 함수를 정의함 !

02. 글로브(GloVe)

1. 동시등장행렬

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

행과 열을 전체 단어 집합의 단어들로 구성하고 i 단어의 윈도우 크기 내에서 k 단어가 등장한 횟수를 i행 k열에 기재한 행렬을 말함

위 행렬은 전치해도 동일한 행렬이 되는 특징이 있다.

02. 글로브(GloVe)

2. 동시등장확률

동시 등장 확률 P(k | i) 는 동시 등장 행렬로부터 특정 단어 i의 전체 등장 횟수를 카운트하고, 특정 단어 i가 등장했을 때 어떤 단어 k가 등장한 횟수를 카운트하여 계산한 조건부 확률입니다.

(i : 중심 단어, k : 주변 단어)

$$P(k \mid i) = \frac{\text{i행 k열의 값}}{\text{중심 단어 i의 행의 모든 값을 더한 값}}$$

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

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

02. 글로브(GloVe)

3. 손실 함수

목적 : 특정 단어 k가 주어졌을 때 임베딩된 두 단어벡터의 내적이 두 단어의 동시등장확률 간 비율이 되게끔 임베딩

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

$$F(w_{ice}, w_{steam}, w_{solid}) = \frac{P_{ice,solid}}{P_{steam,solid}} = \frac{P(solid|ice)}{P(solid|steam)} = \frac{1.9 \times 10^{-4}}{2.2 \times 10^{-5}} = 8.9$$

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

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

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

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

도출된 F의 세가지 특징

1. W_i, W_j 를 바꾸어도 같은 식 반환
2. 대칭 행렬 이므로 F의 성질에 있어야함
3. Homomorphism 조건 만족

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

02. 글로브(GloVe)

3. 손실 함수

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

$$w_i^T \tilde{w}_k = \log P_{ik} = \log X_{ik} - \log X_i$$

문제점)

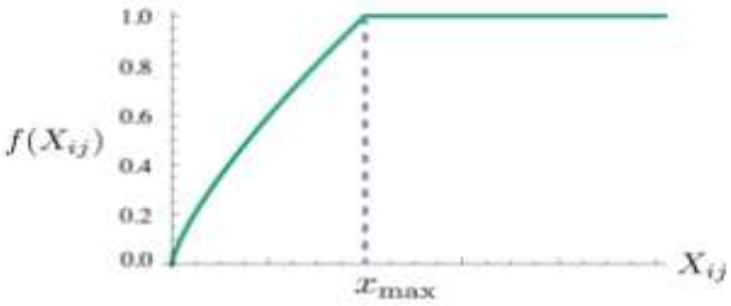
$log(X_{ik}) - log(X_i), log(X_{ki}) - log(X_k)$ 다름

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

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

$$J = \sum_{i,j=1}^{V} (w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log X_{ij})^2$$

우변과 좌변의 차이를 최소로 하는게 d차원 벡터공간에 적절히 임베딩된 단어 벡터들



$$J = \sum_{i,j=1}^{V} f(X_{ij}) (w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log X_{ij})^2$$

$$where \quad f(x) = \begin{cases} (\frac{x}{x_{max}})^{\alpha} \\ 1 \end{cases} \begin{matrix} if \quad x < x_{max} \\ otherwise \end{matrix}$$

02. 글로브(GloVe)

4. 실습

pip install glove_python 설치

```
from glove import Corpus, Glove

corpus = Corpus()
corpus.fit(result, window=5)
# 훈련 데이터로부터 GloVe에서 사용할 동시 등장 행렬 생성

glove = Glove(no_components=100, learning_rate=0.05)
glove.fit(corpus.matrix, epochs=20, no_threads=4, verbose=True)
glove.add_dictionary(corpus.dictionary)
# 학습에 이용할 쓰레드의 개수는 4로 설정, 에포크는 20.
```

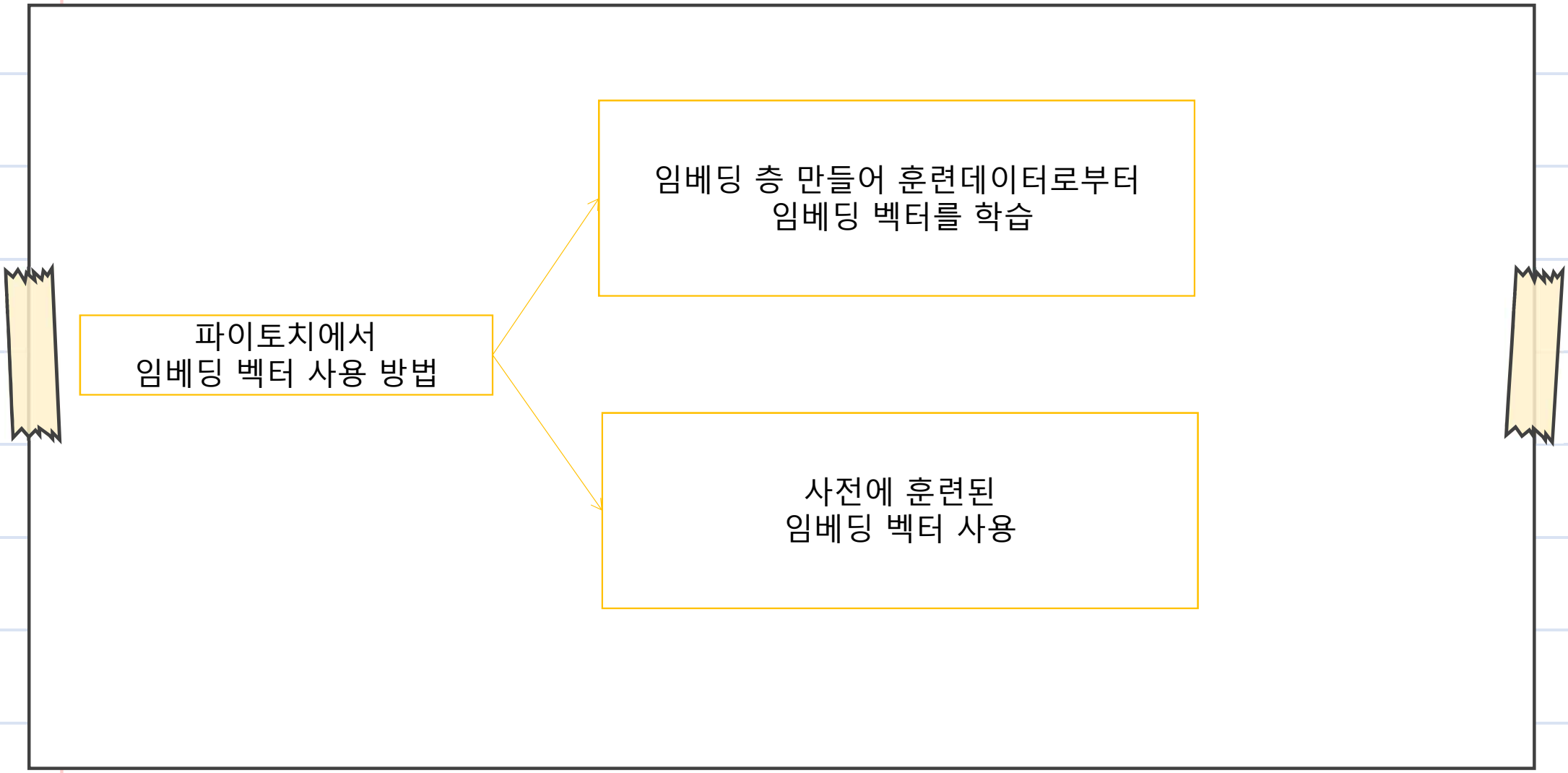
```
model_result3=glove.most_similar("university")
print(model_result3)
```

```
[('harvard', 0.8690162017225468), ('cambridge', 0.8373272000675909), ('mit', 0.8288055170365777), ('stanford', 0.8212712738131419)]
```

```
model_result1=glove.most_similar("man")
print(model_result1)
```

```
[('woman', 0.9621753707315267), ('guy', 0.8860281455579162), ('girl', 0.8609057388487154), ('kid', 0.8383640509911114)]
```

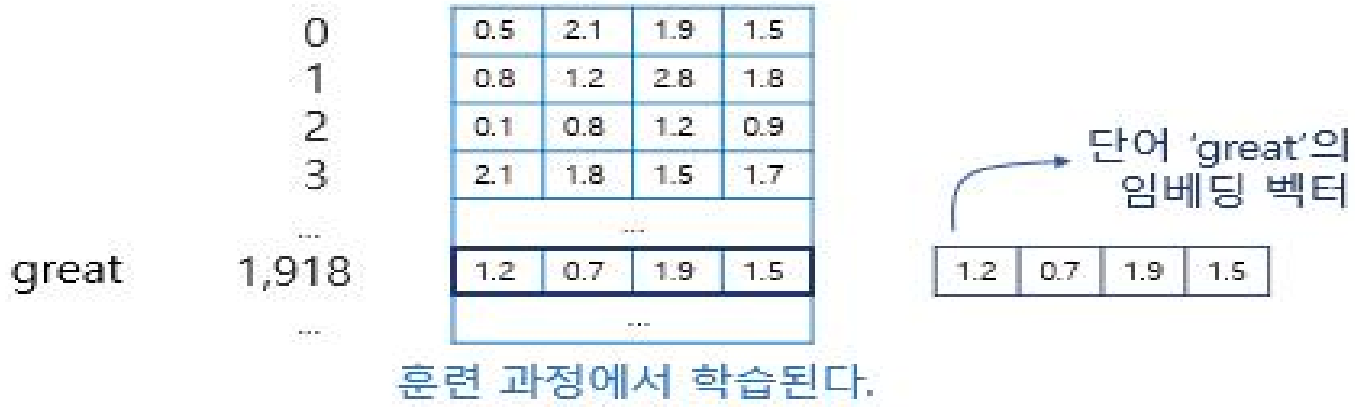
03. 파이토치(PyTorch)의 nn.Embedding()



03. **파이토치**(PyTorch)**의** nn.Embedding()

임베딩 층 만들어 훈련데이터로부터
임베딩 벡터를 학습
=> nn.Embedding() 로 구현

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



03. 파이토치(PyTorch)의 nn.Embedding()

```
train_data = 'you need to know how to code'
word_set = set(train_data.split()) # 중복을 제거한 단어들의 집합인 단어 집합 생성.
vocab = {tkn: i+2 for i, tkn in enumerate(word_set)}  # 단어 집합의 각 단어에 고유한 정수 맵핑.
vocab['<unk>'] = 0
vocab['<pad>'] = 1
```

전처리 과정

```
import torch.nn as nn
embedding_layer = nn.Embedding(num_embeddings = len(vocab),
                               embedding_dim = 3,
                               padding_idx = 1)
```

num_embeddings : 임베딩을 할 단어들의 개수, 단어 집합의 크기.

embedding_dim : 임베딩 할 벡터의 차원.

padding_idx : 선택적으로 사용하는 인자. 패딩을 위한 토큰의 인덱스 알려줌.

```
print(embedding_layer.weight)
```

```
Parameter containing:
tensor([[-0.1778, -1.9974, -1.2478],
        [ 0.0000,  0.0000,  0.0000],
        [ 1.0921,  0.0416, -0.7896],
        [ 0.0960, -0.6029,  0.3721],
        [ 0.2780, -0.4300, -1.9770],
        [ 0.0727,  0.5782, -3.2617],
        [-0.0173, -0.7092,  0.9121],
        [-0.4817, -1.1222,  2.2774]], requires_grad=True)
```

단어 집합의 크기의 행을 가지는 임베딩 테이블 생성

03. 파이토치(PyTorch)의 nn.Embedding()

사전에 훈련된 임베딩 벡터 사용

=> 훈련 데이터가 부족한 상황에서는 훈련되어있는 임베딩 벡터를 불러오는 판단

```
from torchtext import data, datasets
```

```
TEXT = data.Field(sequential=True, batch_first=True, lower=True)
LABEL = data.Field(sequential=False, batch_first=True)
```

```
trainset, testset = datasets.IMDB.splits(TEXT, LABEL)
```

```
print('훈련 데이터의 크기 : {}' .format(len(trainset)))
```

```
훈련 데이터의 크기 : 25000
```

```
print(vars(trainset[0]))
```

```
{'text': ['streisand', 'fans', 'only', 'familiar', 'with', 'her', 'work', 'from', 'the', 'funny', 'girl', 'film', 'onwards',
        ... 중략 ...
        'ever,', 'done', 'anything', 'better!"<br', '/><br', '/>and', 'she', 'was', 'twenty-three', 'years', 'old!'],
'label': 'pos'}
```

03. 파이토치(PyTorch)의 nn.Embedding()

```
from gensim.models import KeyedVectors
```

```
word2vec_model = KeyedVectors.load_word2vec_format('eng_w2v')
```

```
print(word2vec_model['this']) # 영어 단어 'this'의 임베딩 벡터값 출력
```

```
[-0.48601353  2.4053142  -0.8451491  -0.636236   -0.09839931  0.9016884
  ... 중략 ...
  0.06850401  2.321853   -1.2139624  -1.2775816 ]
```

```
print(word2vec_model['self-indulgent']) # 영어 단어 'self-indulgent'의 임베딩 벡터값 출력
```

```
KeyError: "word 'self-indulgent' not in vocabulary"
```

03. 파이토치(PyTorch)의 nn.Embedding()

```
import torch
import torch.nn as nn
from torchtext.vocab import Vectors
```

```
vectors = Vectors(name="eng_w2v") # 사전 훈련된 Word2Vec 모델을 vectors에 저장
```

```
TEXT.build_vocab(trainset, vectors=vectors, max_size=10000, min_freq=10) # Word2Vec 모델을 임베딩 벡터값으로 초기화
```

```
print(TEXT.vocab.stoi)
```

```
{'<unk>': 0, '<pad>': 1, 'the': 2, 'a': 3, 'and': 4, 'of': 5, 'to': 6, 'is': 7, 'in': 8, 'i': 9, 'this': 10, 'that': 11,
... 중략 ...
'seconds.': 9997, 'secure': 9998, 'seeing,': 9999, 'self-indulgent': 10000, 'sequels,': 10001})
```

```
print('임베딩 벡터의 개수와 차원 : {} '.format(TEXT.vocab.vectors.shape))
```

```
임베딩 벡터의 개수와 차원 : torch.Size([10002, 100])
```

```
print(TEXT.vocab.vectors[0]) # <unk>의 임베딩 벡터값
```

```
tensor([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        ... 중략 ...
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

```
print(TEXT.vocab.vectors[1]) # <pad>의 임베딩 벡터값
```

```
tensor([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        ... 중략 ...
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

03. 파이토치(PyTorch)의 nn.Embedding()

```
print(TEXT.vocab.vectors[10]) # this의 임베딩 벡터값
```

```
tensor([-0.4860,  2.4053, -0.8451, -0.6362, -0.0984,  0.9017, -1.8017, -2.4730,
         ... 중략 ...
         0.0685,  2.3219, -1.2140, -1.2776])
```

```
print(TEXT.vocab.vectors[10000]) # 단어 'self-indulgent'의 임베딩 벡터값
```

```
tensor([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        ... 중략 ...
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
        0., 0., 0., 0.])
```

```
embedding_layer = nn.Embedding.from_pretrained(TEXT.vocab.vectors, freeze=False)
```

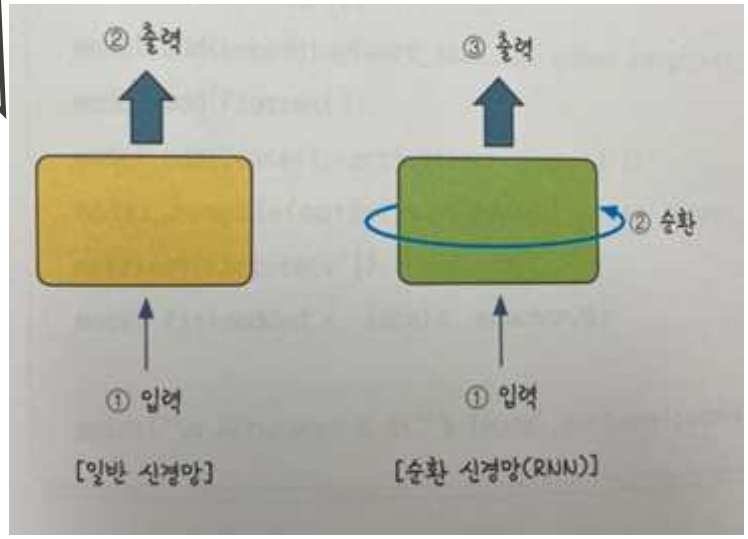
```
print(embedding_layer(torch.LongTensor([10]))) # 단어 this의 임베딩 벡터값
```

```
tensor([[-0.4860,  2.4053, -0.8451, -0.6362, -0.0984,  0.9017, -1.8017, -2.4730,
         ... 중략 ...
         0.0685,  2.3219, -1.2140, -1.2776]], grad_fn=<EmbeddingBackward>)
```

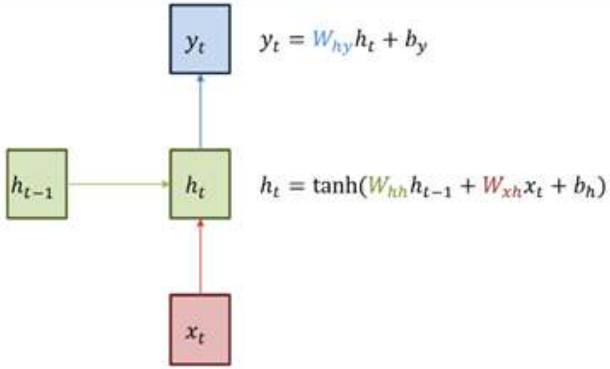
04. 순환신경망(Recurrent Neural Network)

RNN(Recurrent Neural Network)
= 입력과 출력을 시퀀스 단위로 처리하는 모델
= 여러 개의 데이터가 순서대로 입력되었을 때 앞서 입력받은 데이터를 잠시 기억해 놓는 방법

예) 번역기
입력 : 번역하고자 하는 문장. 즉, 단어 시퀀스
출력 : 번역된 문장. 또한 단어 시퀀스



RNN 이란?
히든 노드가 방향을 가진 엣지로 연결돼 순환구조 이룸
-> 순차적으로 등장하는 데이터 처리에 적합한 모델



녹색은 히든 state , 빨강은 인풋, 파랑은 아웃풋
현재 상태의 히든 state Ht는 직전 시점의 히든 state Ht-1를 받아 갱신됩니다.

04. 순환신경망(Recurrent Neural Network)

파이토치의 nn.RNN()

```
import torch
import torch.nn as nn
```

```
input_size = 5 # 입력의 크기
hidden_size = 8 # 은닉 상태의 크기
```

```
# (batch_size, time_steps, input_size)
inputs = torch.Tensor(1, 10, 5)
```

```
cell = nn.RNN(input_size, hidden_size, batch_first=True)
```

```
outputs, _status = cell(inputs)
```

```
print(outputs.shape) # 모든 time-step의 hidden_state
```

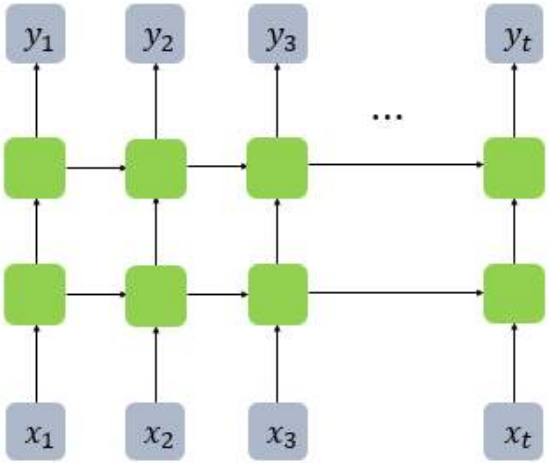
```
torch.Size([1, 10, 8])
```

```
print(_status.shape) # 최종 time-step의 hidden_state
```

```
torch.Size([1, 1, 8])
```

04. 순환신경망(Recurrent Neural Network)

깊은 순환 신경망



은닉층 2개 DRNN

```
# (batch_size, time_steps, input_size)
inputs = torch.Tensor(1, 10, 5)
```

```
cell = nn.RNN(input_size = 5, hidden_size = 8, num_layers = 2, batch_first=True)
```

```
print(outputs.shape) # 모든 time-step의 hidden_state
```

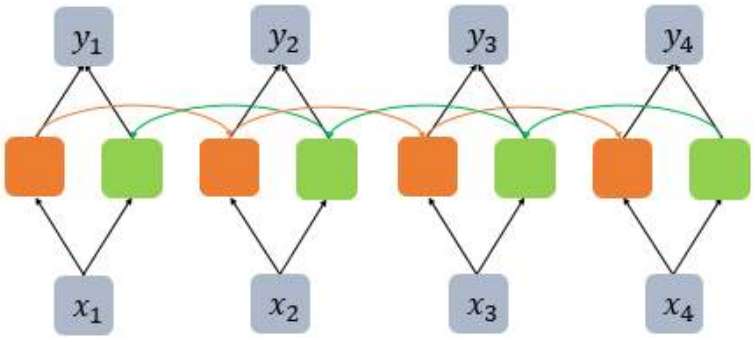
```
torch.Size([1, 10, 8])
```

```
print(_status.shape) # (층의 개수, 배치 크기, 은닉 상태의 크기)
```

```
torch.Size([2, 1, 8])
```

04. 순환신경망(Recurrent Neural Network)

양방향 순환 신경망




```
# (batch_size, time_steps, input_size)
inputs = torch.Tensor(1, 10, 5)
```

```
cell = nn.RNN(input_size = 5, hidden_size = 8, num_layers = 2, batch_first=True, bidirectional = True)
```

```
outputs, _status = cell(inputs)
```

```
print(outputs.shape) # (배치 크기, 시퀀스 길이, 은닉 상태의 크기 x 2)
```

```
torch.Size([1, 10, 16])
```

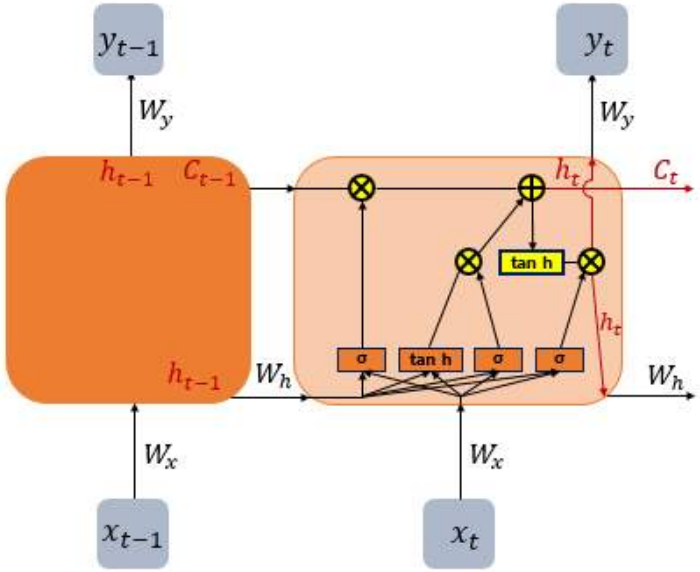
시점 t에서의 출력값을 예측할 때 이전 시점의 데이터뿐만 아니라, 이후 데이터로도 예측할 수 있다.

```
print(_status.shape) # (층의 개수 x 2, 배치 크기, 은닉 상태의 크기)
```

```
torch.Size([4, 1, 8])
```

단방향에 비해 2배가 됨

05. **장단기 메모리**(Long Short-Term Memory)



단점을 보완한 RNN의 일종을 장단기 메모리(Long Short-Term Memory)라고 하며, **LSTM**이라고 함. LSTM은 은닉층의 메모리 셀에 입력 게이트, 망각 게이트, 출력 게이트를 추가하여 불필요한 기억을 지우고, 기억해야할 것들을 정함

긴 시퀀스의 입력을 처리하는데
탁월한 성능을 보임

```
nn.RNN(input_dim, hidden_size, batch_fisrt=True)
```

LSTM 셀은 이와 유사하게 다음과 같이 사용합니다.

```
nn.LSTM(input_dim, hidden_size, batch_fisrt=True)
```

END