# 13. 워드 임베딩

영어에서 '임베드(embed)'는 어딘가에 심다, 끼워넣다 등의 뜻이다. 한 예로 기계장치 등에 소프트웨어를 심어 넣은 것을 임베디드 소프트웨어라고 한다. 워드 임베딩(word embedding)은 단어를 어떤 가상의 공간 상에 한 위치에 심어넣는 것, 또는 그렇게 심어넣은 단어의 좌표를 말한다.

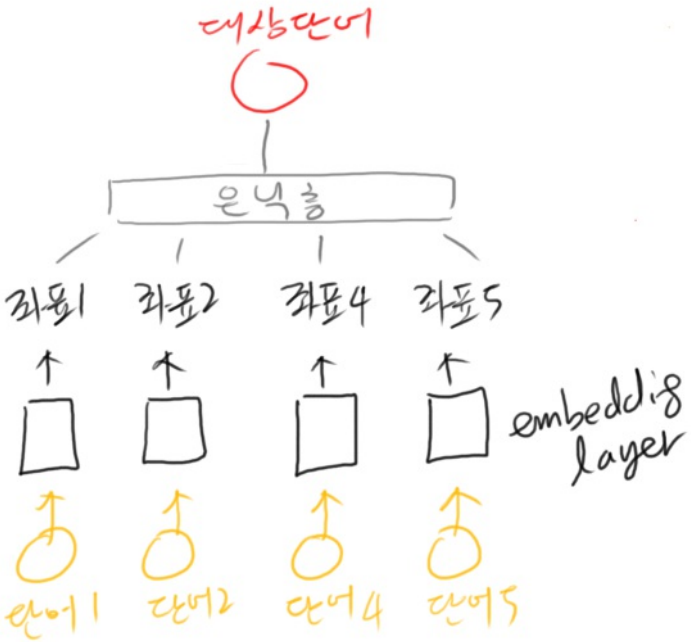
쉽게 생각하면 단어의 의미를 수치화시키는 것으로 생각할 수 있다. 예를 들어 '국회'와 '의회'는 비슷한 의미의 말이다. 컴퓨터는 단어의 의미를 모르기 때문에 두 단어를 전혀 다르게 처리하게 된다. 그러나 워드 임베딩을 통해서 '국회'와 '의회'에 비슷한 좌표를 부여하면 컴퓨터가 좀 텍스트를 잘 분석할 수 있게 된다.

## 13.1. Word2Vec

워드임베딩에는 잠재의미분석, GloVe, Word2Vec 등 다양한 방법들이 있다. 여기서는 Word2Vec에 대해서 알아보기로 한다.

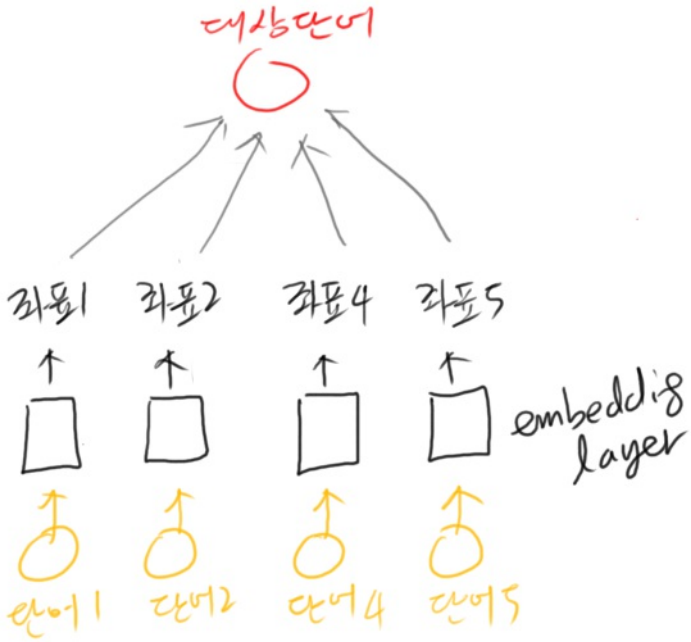
Word2Vec의 시작은 신경망 언어 모형(Neural Network Language Model: 이하 NNLM)이다. NNLM의 기본 원리는 인공신경망을 이용해서 한 단어를 둘러싼 주변의 단어를 이용해서 둘러쌓인 한 단어를 예측하게 하는 것이다.


NNLM의 구조는 평범한 앞먹임 신경망과 같지만 입력층에 단어에 좌표를 부여하는 임베딩 레이어를 넣는다는 점이 다르다.



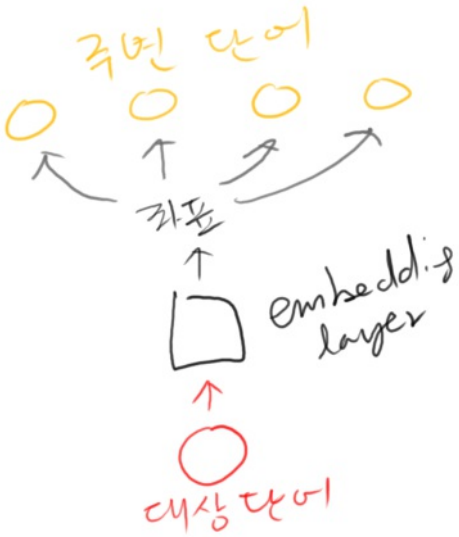

인공신경망을 예측의 오차를 줄이는 방향으로 모든 레이어에서 학습이 이뤄진다. 결과적으로 임베딩 레이어도 주변 단어들이 대상 단어를 가장 잘 예측할 수 있는 좌표를 부여하게 된다.

Word2Vec은 NNLM의 학습 효율을 높인 것이다. Word2Vec에는 CBOW와 Skip-gram 두 종류가 있다. 먼저 CBOW는 NNLM에서 은닉층을 없앤 것이다.




대신 주변단어들의 좌표를 단순히 더해서 대상 단어를 예측한다. 구조가 단순해지므로 더 적은 단어로도 잘 학습시킬 수 있고, 속도도 빨라진다.

Skip-gram은 CBOW를 뒤집어서 대상단어로 주변 단어를 예측하게 한 것이다.




skip-gram으로 학습된 임베딩이 CBOW보다 성능이 더 좋은 편이다. 왜냐하면 CBOW는 여러 단어로 한 단어를 예측하기 때문에 상대적으로 임베딩이 받는 피드백이 적지만, Skip-gram은 여러 단어를 예측하기 때문에 임베딩이 받는 피드백이 많기 때문이다.

## 13.2. Word2Vec

### 13.2.1. 데이터 준비

imdb 데이터셋을 불러온다.

```
from keras.datasets import imdb

(x_train, y_train), (x_test, y_test) = imdb.load_data()
```

단어 번호를 불러온다.

```
word_index = imdb.get_word_index()
```

단어 번호와 단어의 관계를 사전으로 만든다. 1번은 문장의 시작, 2번은 사전에 없는 단어(Out of Vocabulary)로 미리 지정되어 있다.

```
index_word = {idx+3: word for word, idx in word_index.items()}

index_word[1] = '<START>'
index_word[2] = '<UNKNOWN>'
```

단어 번호로 된 데이터를 단어로 변환해 본다. 실제 데이터 분석에서는 단어로 된 데이터를 단어 번호로 변환해야 한다.

```
' '.join(index_word[i] for i in x_train[0])
```

```
"<START> this film was just brilliant casting location scenery story direction everyone's really suited the part they played and you could just
```

단어의 총 갯수를 변수에 할당한다.

```
NUM_WORDS = max(index_word) + 1
```

### 13.2.2. 대상단어와 맥락단어

Word2Vec은 대상 단어와 맥락 단어(대상 단어 주변의 단어)의 관계를 학습시키는 방법이다. 한 단어씩 옮겨가며 좌우로 주변 2단어를 맥락 단어에 포함시키는 함수를 만든다.

```
def extract_target_context(x):
    targets = []  # 대상단어
    contexts = [] # 맥락단어들
    for paragraph in x:
        n = len(paragraph)
        window = 2  # 좌우로 각 2단어씩
        for i in range(window, n-window):
            # 대상단어
            targets.append(paragraph[i])

            # 맥락단어
            contexts.append(
                paragraph[i-window:i] +    # 왼쪽 맥락단어
                paragraph[i+1:i+window+1]) # 오른쪽 맥락단어
    return targets, contexts

target_train, context_train = extract_target_context(x_train)
target_test, context_test = extract_target_context(x_test)
```

데이터에서 첫 5단어는 아래와 같다.

```
x_train[0][:5]
```

```
[1, 14, 22, 16, 43]
```

대상단어는 가운데 22번이고,

```
target_train[0]
```

```
22
```

맥락단어는 왼쪽 1, 14번과 오른쪽 16, 43번이 된다.

```
context_train[0]
```

```
[1, 14, 16, 43]
```

### 13.2.3. 데이터 생성자

데이터를 신경망에 학습시키려면 행렬 형태로 바꿔야 한다. 그러나 데이터가 매우 많기 때문에 한 번에 행렬로 바꾸면 학습시키기가 어렵다. 그래서 생성자라는 형태로 만들어준다.

```
from keras.utils import Sequence, to_categorical

import numpy
```

keras는 Sequence 클래스를 상속하는 형태로 데이터 생성자를 만들 수 있도록 하고 있다.

```
class TargetContext(Sequence):
    def __init__(self, target, context, batch_size):
        self.target = numpy.asarray(target)
        self.context = numpy.asarray(context)
        self.batch_size = batch_size

    def __len__(self):
        """데이터의 길이"""
        # return len(self.context) // self.batch_size
        return 128  # 간단히 하기 위해 128로 고정

    def __getitem__(self, idx):
        """idx 번째 데이터 배치를 가져온다"""
        i = numpy.random.choice(len(self.target), self.batch_size)  # 실제로는 무작위로 batch_size만큼 데이터를 고른다
        batch_x = self.context[i]
        batch_y = self.target[i]
        return batch_x, to_categorical(batch_y, NUM_WORDS)  # to_categorical로 y를 one-hot encoding을 한다

train = TargetContext(target_train, context_train, 32)
valid = TargetContext(target_test, context_test, 32)

train[0]
```

```
(array([[ 3620,   463,    89,   363],
        [  483,  4051,  3781,  5861],
        [ 6465,  2465,    56,    18],
        [ 9442,     5,   265,    29],
        [   23,    27,    21,   444],
        [  252,    48,   276,    11],
        [   10,    10,   147,   649],
        [  124,   625,   326,    12],
        [    8,   140,     4,  2019],
        [    7,   148,   162,  5097],
        [  642,  2373,  5789,    19],
        [   39,     4,   559,     7],
        [  856,   232, 85225,    52],
        [   55,  1288,    49,   680],
        [18111,    11,     7,  6663],
        [  105,   121,    19,    46],
        [  216,   125,   916,     4],
        [    8,    72, 66504,  1294],
        [    4,   543,    28,    77],
        [   27,   649,    27,   452],
        [  144,    30, 54831,   328],
        [  598,  1310, 74593,    13],
        [ 4405,   178,     6,  2532],
        [  127,    59,   142,    40],
        [13362,  5487,   180,   263],
        [   16,   626,    11,     4],
        [   27, 11731,    27,   980],
        [ 3548,  2104,     6,  2969],
        [   12,  1160,   185,   250],
```

```
        [   10,  1174,     6,   947],
        [ 1584,   115,   439,    11],
        [    8,    30,     8,   714]]), array([[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., 0., ..., 0., 0., 0.],
        [0., 0., 0., ..., 0., 0., 0.]], dtype=float32))
```

### 13.2.4. NNLM

```
from keras.models import Sequential

from keras.layers import Dense, Embedding, Flatten

from keras.optimizers import Adam

from keras.callbacks import EarlyStopping
```

임베딩 레이어는 저장을 위해 따로 변수로 지정해둔다.

```
emb_nnlm = Embedding(input_dim=NUM_WORDS, output_dim=8, input_length=4)

nnlm = Sequential()
nnlm.add(emb_nnlm)
nnlm.add(Flatten())
nnlm.add(Dense(128, activation='relu'))
nnlm.add(Dense(NUM_WORDS, activation='softmax'))

nnlm.summary()

nnlm.compile(optimizer=Adam(), loss='categorical_crossentropy', metrics=['accuracy'])

nnlm.fit_generator(
    train,
    epochs=30,
    validation_data=valid,
    callbacks=[EarlyStopping(monitor='val_acc')])
```

```
<keras.callbacks.History at 0x7ff55164df28>
```

#### 임베딩 레이어 저장

```
numpy.save('emb_nnlm.npy', emb_nnlm.get_weights()[0])
```

### 13.2.5. CBOW

CBOW는 NNLM에서 은닉층을 없애고 대신 임베딩을 단순히 평균낸 것을 사용한다.

```
from keras.layers import Lambda
from keras import backend as K

emb_cbow = Embedding(input_dim=NUM_WORDS, output_dim=8, input_length=4)

cbow = Sequential()
cbow.add(emb_cbow)
cbow.add(Lambda(lambda x: K.mean(x, axis=1)))  # 임베딩의 평균
cbow.add(Dense(NUM_WORDS, activation='softmax'))

cbow.summary()
```

#### 학습

```
cbow.compile(optimizer=Adam(), loss='categorical_crossentropy', metrics=['accuracy'])

cbow.fit_generator(
    train,
    epochs=30,
    validation_data=valid,
    callbacks=[EarlyStopping(monitor='val_acc')])
```

```
<keras.callbacks.History at 0x7ff55f1a9da0>
```

#### 임베딩 레이어 저장

```
numpy.save('emb_cbow.npy', emb_cbow.get_weights()[0])
```

### 13.2.6. Skip-gram

Skipgram은 구현하기가 조금 까다롭다. 맥락단어를 직접 예측하는 대신 대상 단어와 맥락 단어가 함께 입력되면 1을 출력하고, 대상 단어와 맥락 외 단어가 함께 입력되면 0을 출력하는 형식으로 만든다.

```
class SkipGramData(TargetContext):
    def __getitem__(self, idx):
        n = self.batch_size // 2
        i = numpy.random.choice(len(self.target), n)
        j = numpy.random.choice(4, n)

        true_context = self.context[i, j]  # 실제 맥락
        true_target = self.target[i]       # 실제 대상
        true_y = numpy.ones(n)             # 1

        # 무작위로 단어를 뽑아 가짜 맥락을 만든다
        false_context = numpy.random.choice(NUM_WORDS, n)
        false_y = numpy.zeros(n)                          # 0

         # 실제 맥락과 가짜 맥락을 이어 붙인다
        context = numpy.append(true_context, false_context)

        # 실제 대상을 2배로 한다
        target = numpy.append(true_target, true_target)
```

```
        # 앞부분은 1, 뒷부분은 0
        y = numpy.append(true_y, false_y)

        return [context, target], y

train_skipgram = SkipGramData(target_train, context_train, 32)
valid_skipgram = SkipGramData(target_test, context_test, 32)
```

모형은 이제까지 사용한 `Sequential` 모형 대신 `keras`의 함수형 방식을 사용한다. 이 방식은 각 레이어를 일종의 함수처럼 쓰는 방법이다.

```
from keras.layers import Activation, Dot, Input, Reshape

from keras.models import Model

# 입력 레이어
input_target = Input(shape=(1,))
input_context = Input(shape=(1,))

# 임베딩 레이어
emb_skip_target = Embedding(input_dim=NUM_WORDS, output_dim=8)
emb_skip_context = Embedding(input_dim=NUM_WORDS, output_dim=8)

# 입력 레이어를 임베딩 레이어에 연결시키고
# 양쪽 임베딩 레이어를 Dot 레이어에 연결시킨다
# Dot 레이어는 양쪽 입력을 곱하는 역할을 한다
out = Dot(axes=2)([
    emb_skip_target(input_target),
    emb_skip_context(input_context)])

# 출력 형태를 (1, 1)에서 (1,)로 바꾼다
out = Reshape((1,), input_shape=(1, 1))(out)

# 시그모이드로 출력한다
out = Activation('sigmoid')(out)

skipgram = Model(inputs=[input_target, input_context], outputs=out)

skipgram.summary()

skipgram.compile(optimizer=Adam(), loss='binary_crossentropy', metrics=['accuracy'])

skipgram.fit_generator(
    train_skipgram,
    epochs=30,
    validation_data=valid_skipgram,
    callbacks=[EarlyStopping(monitor='val_acc')])
```

```
<keras.callbacks.History at 0x7ff55ec48630>
```

#### 임베딩 레이어 저장

```
numpy.save('emb_skip_target.npy', emb_skip_target.get_weights()[0])
```

### 13.2.7. 임베딩 레이어 재사용

워드 임베딩을 학습시키는 이유는 그 자체로 목적이 있다기보다 다른 학습에 이를 재사용하여 학습 효율을 높이기 위해서다.

```
from keras.preprocessing.sequence import pad_sequences
from keras.utils import to_categorical

MAXLEN = 20
```

저장한 레이어의 가중치를 불러온다.

```
w = numpy.load('emb_skip_target.npy')
```

임베딩 레이어를 만든다. 아래에서 `trainable=False`로 하면 임베딩 레이어는 추가 학습을 하지 않는다.

```
emb_ff = Embedding(input_dim=NUM_WORDS, output_dim=8, input_length=MAXLEN,
                   weights=[w],   # 레이어 가중치를 저장한 값으로 설정한다
                   trainable=True)
```

#### 앞먹임 신경망

RNN 수업에서 사용했던 앞먹임 신경망을 다시 만들어보자.

```
x_train_seq = pad_sequences(x_train, MAXLEN)
x_test_seq = pad_sequences(x_test, MAXLEN)

ff = Sequential()
ff.add(emb_ff)  # 미리 만들어진 임베딩 레이어를 사용한다.
ff.add(Flatten())
ff.add(Dense(1, activation='sigmoid'))

ff.summary()

from keras.optimizers import RMSprop

ff.compile(optimizer=RMSprop(), loss='binary_crossentropy', metrics=['acc'])

ff.fit(
  x_train_seq,
    y_train,
    epochs=30,
    batch_size=128,
    validation_split=0.2,
    callbacks=[EarlyStopping(monitor='val_acc')])
```

```
<keras.callbacks.History at 0x7ff55e454240>
```