단어 임베딩 사용하기

- 단어와 벡터를 연관짓는 강력하고 인기 있는 또 다른 방법 중의 하나는 단어 임베딩이라는 밀집 단어 벡터를 사용하는 것.

01 Onehot vs 단어 임베딩 비교

- 고차원 저차원
  - 원핫 인코딩 기법으로 만든 벡터는 **대부분 0으로 채워지는 고차원**이다.
  - 단어 임베딩은 **저차원 기법**이다.
  - 원핫인코딩은 단어 사전이 2만개의 토큰으로 이루어져 있다면 20,000차원의 벡터를 사용
  - 보통 단어 임베딩은 256, 512, 1024차원의 단어 임베딩을 사용.

02 단어 임베딩 만드는 두가지 방법

- 신경망을 학습하는 방법처럼 단어 벡터를 학습하기
- 사전에 훈련된 단어 임베딩을 로드하기(pretrained word embedding)
  - 구글의 토마스 미코로프 word2vec 알고리즘 (https://code.google.com/archive/p/word2vec)
  - 스탠포드 대학교 : GloVe(https://nlp.stanford.edu/projects/glove)

03 Embedding 층을 사용하여 단어 임베딩 학습하기

- 단어와 밀집 벡터를 연관 짓는 가장 간단한 방법은 랜덤하게 벡터를 선택하는 것.
- 이 방식의 문제점은 임베딩 공간이 구조적이지 않다는 것.
  - 예를 들어, accurate와 exact단어는 대부분 문장에서 비슷한 의미로 사용. 단, 다른 임베딩을 갖는다.
  - 심층 신경망이 이런 임의의 구조적이지 않은 임베딩 공간을 이해하기는 어렵다.

그러면 어떻게 해야 할까?

- 단어 벡터 사이에 좀 더 추상적이고 기하학적인 관계를 얻으려면 단어 사이에 있는 의미 관계를 반영해야함.
  - 단어 임베딩은 언어를 기하학적 공간에 매핑하는 것.
  - 예를 들어 잘 구축된 임베딩 공간에서는 동의어가 비슷한 단어 벡터로 임베딩된다.
  - 일반적으로 두 단어 벡터 사이의 거리(L2 거리)는 이 단어 사이의 의미 거리와 관계되어 있다.
    - 멀리 떨어진 위치에 임베딩된 단어 의미는 서로 다른 반면
    - 비슷한 단어들은 가까이 임베딩된다.

문제는 사람의 언어를 완전히 매핑시킬 수 있는 이상적인 단어 임베딩 공간을 만드는 것이다. 이런 공간이 있을까?

케라스를 이용하여 이를 구현해 보자

- Embedding 층(특정 단어를 나타내는) 정수 인덱스를 밀집 벡터로 매핑하는 딕셔너리로 이해
- 정수를 입력받아, 내부 딕셔너리에서 이 정수와 연관된 벡터를 찾아 반환

In [1]:
```
from keras.layers import Embedding

# Embedding 층은 적어도 두 개의 매개변수를 사용.
# 가능한 토큰의 개수(여기서는 1,000으로 단어 인덱스 최댓값 + 1입니다)와
# 임베딩 차원(여기서는 64)입니다
embedding_layer = Embedding(1000, 64)
```

Embedding층의 입력

- (samples, sequence_length)
  - samples : 샘플수
  - sequence_length : 시퀀스 길이
  - 정수 텐서를 입력으로 받음. 2D텐서
- 여기서 sequence_length가 작은 길이의 시퀀스는 0으로 패딩되고, 긴 시퀀스는 잘리게 됩니다.

Embedding층의 출력

- (samples, sqeuence_length, embedding_dimensionality)
  - samples : 샘플수
  - sequence_length : 시퀀스 길이
  - embedding_dimensionality : 임베딩 차원
  - 출력은 3D 텐서가 된다.

Embedding 층의 객체

- 가중치는 다른 층과 마찬가지로 랜덤하게 초기화
- 신경망의 학습을 통해 점차 조정되어진다.
  - 훈련이 끝나면 임베딩 공간은 특정 문제에 특화된 구조를 많이 갖는다.

04 IMDB 데이터를 이용한 Embedding 실습

In [2]:
```
from keras.datasets import imdb
from keras import preprocessing
```

In [3]:
```
# 특성으로 사용할 단어의 수
max_features = 10000

# 정수 리스트로 데이터를 로드합니다.
(X_train, y_train), (X_test, y_test) = imdb.load_data(num_words=max_features)
```

```
<__array_function__ internals>:5: VisibleDeprecationWarning: Creating an ndarray from
ragged nested sequences (which is a list-or-tuple of lists-or-tuples-or ndarrays with
different lengths or shapes) is deprecated. If you meant to do this, you must specify
'dtype=object' when creating the ndarray
c:\Users\front\anaconda3\envs\tf2x\lib\site-packages\tensorflow\python\keras\datasets
\imdb.py:159: VisibleDeprecationWarning: Creating an ndarray from ragged nested sequen
ces (which is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths or
shapes) is deprecated. If you meant to do this, you must specify 'dtype=object' when c
reating the ndarray
```

```
  x_train, y_train = np.array(xs[:idx]), np.array(labels[:idx])
c:₩users₩front₩anaconda3₩envs₩tf2x₩lib₩site-packages₩tensorflow₩python₩keras₩datasets
₩imdb.py:160: VisibleDeprecationWarning: Creating an ndarray from ragged nested sequen
ces (which is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths or
shapes) is deprecated. If you meant to do this, you must specify 'dtype=object' when c
reating the ndarray
  x_test, y_test = np.array(xs[idx:]), np.array(labels[idx:])
```

In [4]:
```
X_train.shape, y_train.shape, X_test.shape, y_test.shape
```

Out[4]:
```
((25000,), (25000,), (25000,), (25000,))
```

In [7]:
```
# 리뷰의 길이와 10개 단어(인덱스) 보기
len(X_train[0]), X_train[0][0:10]
```

Out[7]:
```
(218, [1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65])
```

다양한 길이가 리뷰가 있을 것이다.

- 리뷰에서 맨 마지막 50개 단어를 얻고, 나머지는 버린다. 또는 길이가 짧다면 0으로 채운다.

In [8]:
```
# 리스트를 (samples, maxlen) 크기의 2D 정수 텐서로 변환합니다.
maxlen = 50

X_train_n = preprocessing.sequence.pad_sequences(X_train, maxlen=maxlen)
X_test_n = preprocessing.sequence.pad_sequences(X_test, maxlen=maxlen)
```

In [9]:
```
X_train.shape, X_test.shape, X_train_n.shape, X_test_n.shape
```

Out[9]:
```
((25000,), (25000,), (25000, 50), (25000, 50))
```

왜 1D 텐서를 2D텐서로 변경하는가?

- Embedding() 층은 다음과 같이 입력을 받는다.
  - (samples, sequence_length)

In [10]:
```
from keras.models import Sequential
from keras.layers import Flatten, Dense, Embedding
```

In [15]:
```
model = Sequential()

# 나중에 임베딩된 입력을 Flatten 층에서 펼치기 위해 Embedding 층에
# input_length를 지정
# Embedding 층의 입력은 2D (samples, maxlen) 이다.
model.add(Embedding(10000, 16, input_length=maxlen))
# Embedding 층의 출력 크기는 (samples, maxlen, 8)가 됩니다.
```

- 무슨 말인가?
  - 50개의 단어를 학습을 통해 16차원 임베딩 공간을 만든다는 것이다.

In [16]:
```
# 3D 임베딩 텐서를 (samples, maxlen * 8) 크기의 2D 텐서로 펼칩니다.
model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))
```

```
# 분류기를 추가합니다.
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])

model.summary()

history = model.fit(X_train_n, y_train,
                    epochs=10,
                    batch_size=32,
                    validation_split=0.2)
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_3 (Embedding)      (None, 50, 16)            160000
_________________________________________________________________
flatten_2 (Flatten)          (None, 800)               0
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 801
=================================================================
Total params: 160,801
Trainable params: 160,801
Non-trainable params: 0
_________________________________________________________________
Epoch 1/10
625/625 [==============================] - 2s 2ms/step - loss: 0.6738 - acc: 0.5963 -
val_loss: 0.5053 - val_acc: 0.7730
Epoch 2/10
625/625 [==============================] - 1s 2ms/step - loss: 0.4402 - acc: 0.8120 -
val_loss: 0.4120 - val_acc: 0.8048
Epoch 3/10
625/625 [==============================] - 1s 2ms/step - loss: 0.3421 - acc: 0.8562 -
val_loss: 0.4029 - val_acc: 0.8146
Epoch 4/10
625/625 [==============================] - 1s 2ms/step - loss: 0.2939 - acc: 0.8757 -
val_loss: 0.3966 - val_acc: 0.8178
Epoch 5/10
625/625 [==============================] - 1s 2ms/step - loss: 0.2637 - acc: 0.8931 -
val_loss: 0.4021 - val_acc: 0.8172
Epoch 6/10
625/625 [==============================] - 1s 2ms/step - loss: 0.2272 - acc: 0.9105 -
val_loss: 0.4103 - val_acc: 0.8168
Epoch 7/10
625/625 [==============================] - 1s 2ms/step - loss: 0.2024 - acc: 0.9246 -
val_loss: 0.4184 - val_acc: 0.8144
Epoch 8/10
625/625 [==============================] - 1s 2ms/step - loss: 0.1692 - acc: 0.9401 -
val_loss: 0.4308 - val_acc: 0.8120
Epoch 9/10
625/625 [==============================] - 1s 2ms/step - loss: 0.1395 - acc: 0.9560 -
val_loss: 0.4483 - val_acc: 0.8064
Epoch 10/10
625/625 [==============================] - 1s 2ms/step - loss: 0.1163 - acc: 0.9665 -
val_loss: 0.4642 - val_acc: 0.8030
```

- 검증 정확도가 80.3%입니다.
- 리뷰 50개의 단어만 사용하여 좋은 결과를 얻었습니다.
- 임베딩 층을 펼쳐 하나의 Dense 층으로 훈련하였으므로 입력 시퀀스 있는 각 단어를 독립적으로 다루었습니다.
- 단어 사이의 관계나 문장 구조를 고려하지 않음.
  - 해결책 : 각 시퀀스 전체를 고려한 특성이 학습되도록 **임베딩 층 위에 순환 층이나 1D 합성곱 층을 추가**하는 것이 좋다.

토큰을 벡터로 변환하는 방법

- 원-핫 인코딩
- 원-핫 해싱 : 각 단어에 명시적으로 인덱스를 할당. 임의의 사이즈에 데이터를 매핑.
- 단어 임베딩 사용
  - 케라스에서 Embedding을 이용하여 일정단어를 일정 차원의 수로 단어를 벡터화 시킨다.
  - 여기서의 가중치는 학습을 통해 특정 데이터에 특정된 임베딩 공간이 만들어진다.

In [ ]: