단어 임베딩 사용하기

학습 내용

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

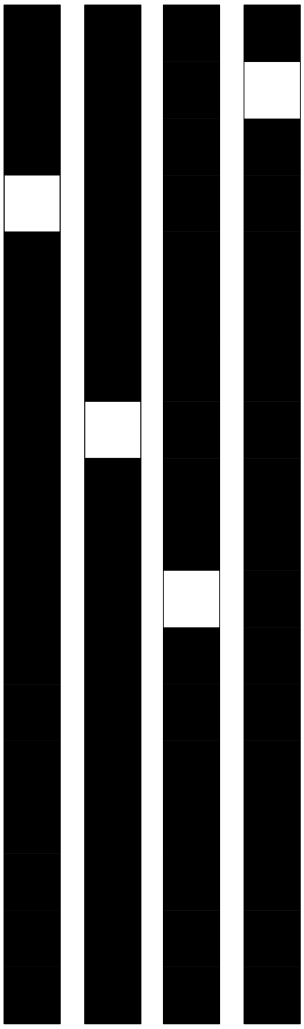
01 Onehot vs 단어 임베딩 비교

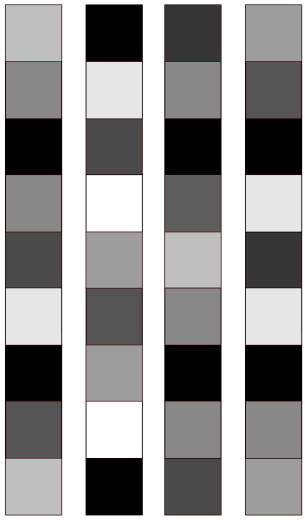
- 고차원 저차원
  - 원핫 인코딩 기법으로 만든 벡터는 대부분 0으로 채워지는 고차원이다.
  - 단어 임베딩은 저차원 기법이다.
- 원핫인코딩은 단어 사전이 2만개의 토큰으로 이루어져 있다면 20,000차원의 벡터를 사용
- 보통 단어 임베딩은 256, 512, 1024차원의 단어 임베딩을 사용.

02 단어 임베딩 만드는 두가지 방법

- 신경망을 학습하는 방법처럼 단어 벡터를 학습하기
- 사전에 훈련된 단어 임베딩을 로드하기(pretrained word embedding)

03 케라스를 활용한 정수 인덱스를 밀집 벡터로 매핑하기(단어 임베딩)





One-hot word vectors:
- Sparse
- High-dimensional
- Hard-coded

Word embeddings:
- Dense
- Lower-dimensional
- Learned from data

- 새로운 작업에는 새로운 임베딩을 학습하는 것이 맞음.
- 다행히 케라스를 사용하면 더 쉽게 학습이 가능 `Embedding` 층의 가중치를 학습

In [3]:

```
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) 정수 텐서를 입력으로 받음. 2D텐서
  - 여기서 sequence_length가 작은 길이의 시퀀스는 0으로 패딩되고, 긴 시퀀스는 잘리게 됩니다.
- Embedding층의 출력 : (samples, sqeuence_length, embedding_dimensionality) 3D텐서
  - 3D텐서는 RNN층이나 1D 합성곱 층에서 처리.

- Embedding층의 가중치(토큰 벡터를 위한 내부 딕셔너리)는 다른 층과 마찬가지로 랜덤하게 초기화

04. IMDB데이터에 대한 Embedding층의 활용과 신경망 분류

In [24]:

```
from keras.datasets import imdb
from keras import preprocessing

# 특성으로 사용할 단어의 수
max_features = 10000

# 사용할 텍스트의 길이(가장 빈번한 max_features 개의 단어만 사용합니다)
maxlen = 20

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

In [25]:

```
X_train.shape, y_train.shape, X_test.shape, y_test.shape
```

Out[25]:

```
((25000,), (25000,), (25000,), (25000,))
```

In [29]:

```
X_train[0]
```

Out[29]:

```
[1,
 14,
 22,
 16,
 43,
 530,
 973,
 1622,
 1385,
 65,
 458,
 4468,
 66,
 3941,
 4,
 173,
 36,
```

- 리뷰에서 맨 마지막 20개 단어를 얻고, 나머지는 버린다.

In [27]:

```
# 리스트를 (samples, maxlen) 크기의 2D 정수 텐서로 변환합니다.
X_train_n = preprocessing.sequence.pad_sequences(X_train, maxlen=maxlen)
X_test_n = preprocessing.sequence.pad_sequences(X_test, maxlen=maxlen)

X_train.shape, X_test.shape
```

Out[27]:

```
((25000,), (25000,))
```

In [28]:

```
X_train_n[0]
```

Out[28]:

```
array([  65,   16,   38, 1334,   88,   12,   16,  283,    5,   16, 4472,
       113,  103,   32,   15,   16, 5345,   19,  178,   32])
```

In [30]:

```
from keras.models import Sequential
from keras.layers import Flatten, Dense, Embedding
```

In [32]:

```
model = Sequential()

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

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

In [36]:

```
X_train_n.shape, y_train.shape
```

Out[36]:

```
((25000, 20), (25000,))
```

In [35]:

```
# 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_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_3 (Embedding)      (None, 20, 8)             80000
_________________________________________________________________
flatten (Flatten)            (None, 160)               0
_________________________________________________________________
dense (Dense)                (None, 1)                 161
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 2
=================================================================
Total params: 80,163
Trainable params: 80,163
Non-trainable params: 0
_________________________________________________________________
Epoch 1/10
625/625 [==============================] - 2s 2ms/step - loss: 0.6944 - acc: 0.5036
- val_loss: 0.6829 - val_acc: 0.5822
Epoch 2/10
625/625 [==============================] - 1s 2ms/step - loss: 0.6684 - acc: 0.6517
- val_loss: 0.6358 - val_acc: 0.7088
Epoch 3/10
625/625 [==============================] - 1s 2ms/step - loss: 0.6091 - acc: 0.7543
- val_loss: 0.5911 - val_acc: 0.7358
Epoch 4/10
625/625 [==============================] - 1s 2ms/step - loss: 0.5531 - acc: 0.7882
- val_loss: 0.5593 - val_acc: 0.7448
Epoch 5/10
625/625 [==============================] - 1s 2ms/step - loss: 0.5044 - acc: 0.8104
- val_loss: 0.5378 - val_acc: 0.7490
Epoch 6/10
625/625 [==============================] - 1s 2ms/step - loss: 0.4695 - acc: 0.8189
- val_loss: 0.5227 - val_acc: 0.7546
Epoch 7/10
625/625 [==============================] - 1s 2ms/step - loss: 0.4412 - acc: 0.8291
- val_loss: 0.5163 - val_acc: 0.7544
Epoch 8/10
625/625 [==============================] - 1s 2ms/step - loss: 0.4088 - acc: 0.8439
- val_loss: 0.5138 - val_acc: 0.7550
Epoch 9/10
625/625 [==============================] - 1s 2ms/step - loss: 0.3906 - acc: 0.8505
- val_loss: 0.5155 - val_acc: 0.7554
Epoch 10/10
625/625 [==============================] - 1s 2ms/step - loss: 0.3684 - acc: 0.8594
- val_loss: 0.5191 - val_acc: 0.7524
```

In [ ]: