KERAS TEXT DATA USE

문서 분류

감성 분석

텍스트는 가장 흔한 시퀀스 형태의 데이터

저자 식별

질의 응답

텍스트 벡터화: 텍스트를 수치형 텐서로 변환하는 과정

텍스트를 단어로 나누고 각 단어를 하나의 벡터로 변환

텍스트를 문자로 나누고 각 문자를 하나의 벡터로 변환

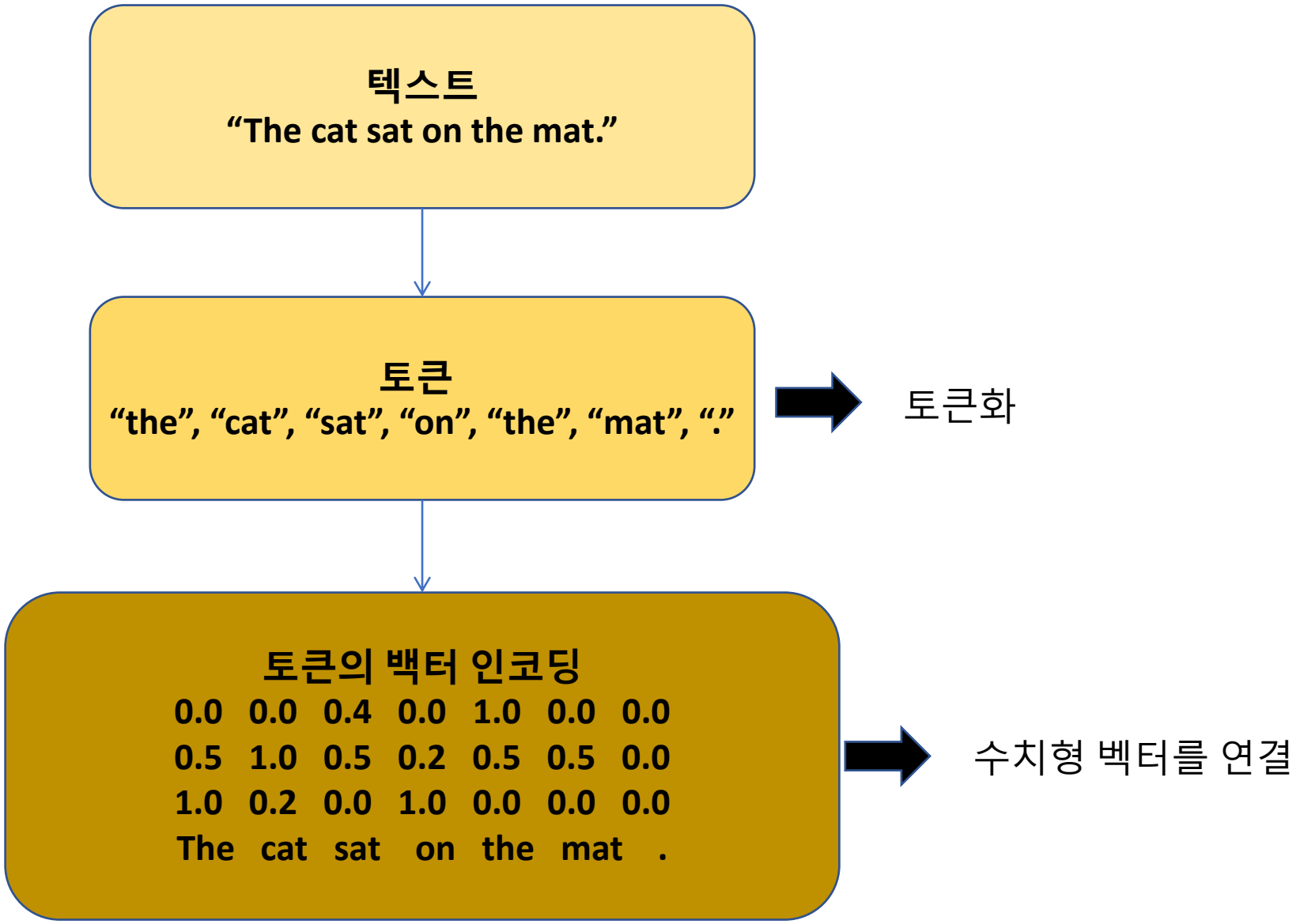
텍스트에서 단어나 문자의 N-그램을 추출하여 각 **N-그램**을 하나의 벡터로 변환.

문장에서 추출한 N개(또는 그 이하)의 연속된 단어 그룹

N-그램은 연속된 단어나 문자의 그룹으로 텍스트에서 단어나 문자를 하나씩 이동하면서 추출

토큰: 텍스트를 나누는 단위(단어, 문자, N-그램)

토큰화: 텍스트를 토큰으로 나누는 작업


텍스트
“The cat sat on the mat.”
토큰
“the”, “cat”, “sat”, “on”, “the”, “mat”, “.”
토큰화
토큰의 백터 인코딩
0.0 0.0 0.4 0.0 1.0 0.0 0.0
0.5 1.0 0.5 0.2 0.5 0.5 0.0
1.0 0.2 0.0 1.0 0.0 0.0 0.0
The cat sat on the mat .
수치형 벡터를 연결

원-핫 인코딩은 토큰을 벡터로 변환하는 가장 일반적이고 기본적인 방법

EX) Cat is cute.

Cat 1 0 0 0

Is 0 1 0 0

Cute 0 0 1 0

. 0 0 0 1

단어 수준의 원-핫 인코딩 =>

```python
import numpy as np

# 초기 데이터: 각 원소가 샘플입니다
# (이 예에서 하나의 샘플이 하나의 문장입니다, 하지만 문서 전체가 될 수도 있습니다)
samples = ['The cat sat on the mat.', 'The dog ate my homework.']

# 데이터에 있는 모든 토큰의 인덱스를 구축합니다
token_index = {}
for sample in samples:
    # split() 메서드를 사용해 샘플을 토큰으로 나눕니다.
    # 실전에서는 구둣점과 특수 문자도 사용합니다.
    for word in sample.split():
        if word not in token_index:
            # 단어마다 고유한 인덱스를 할당합니다.
            token_index[word] = len(token_index) + 1
            # 인덱스 0은 사용하지 않습니다.

# 샘플을 벡터로 변환합니다.
# 각 샘플에서 max_length 까지 단어만 사용합니다.
max_length = 10

# 결과를 저장할 배열입니다
results = np.zeros((len(samples), max_length, max(token_index.values()) + 1))
for i, sample in enumerate(samples):
    for j, word in list(enumerate(sample.split()))[:max_length]:
        index = token_index.get(word)
        results[i, j, index] = 1.
```

문자 수준 원-핫 인코딩(간단한 예)

```
import string

samples = ['The cat sat on the mat.', 'The dog ate my homework.']
characters = string.printable  # 출력 가능한 모든 아스키(ASCII) 문자
token_index = dict(zip(characters, range(1, len(characters) + 1)))

max_length = 50
results = np.zeros((len(samples), max_length, max(token_index.values()) + 1))
for i, sample in enumerate(samples):
    for j, character in enumerate(sample[:max_length]):
        index = token_index.get(character)
        results[i, j, index] = 1.
```

10진	16진	문자
96	0x60	`
97	0x61	a
98	0x62	b
99	0x63	c
100	0x64	d
101	0x65	e
102	0x66	f
103	0x67	g
104	0x68	h
105	0x69	i
106	0x6A	j
107	0x6B	k
108	0x6C	l
109	0x6D	m
110	0x6E	n
111	0x6F	o
112	0x70	p
113	0x71	q
114	0x72	r
115	0x73	s
116	0x74	t

<케라스를 이용한 단어 수준의 원-핫 인코딩>

```
from keras.preprocessing.text import Tokenizer

samples = ['The cat sat on the mat.', 'The dog ate my homework.']

# 가장 빈도가 높은 1,000개의 단어만 선택하도록 Tokenizer 객체를 만듭니다.
tokenizer = Tokenizer(num_words=1000)
# 단어 인덱스를 구축합니다.
tokenizer.fit_on_texts(samples)

# 문자열을 정수 인덱스의 리스트로 변환합니다.
sequences = tokenizer.texts_to_sequences(samples)

# 직접 원-핫 이진 벡터 표현을 얻을 수 있습니다.
# 원-핫 인코딩 외에 다른 벡터화 방법들도 제공합니다!
one_hot_results = tokenizer.texts_to_matrix(samples, mode='binary')

# 계산된 단어 인덱스를 구합니다.
word_index = tokenizer.word_index
print('Found %s unique tokens.' % len(word_index))
```

The cat sat on the mat.
1 4 7 2 6 5

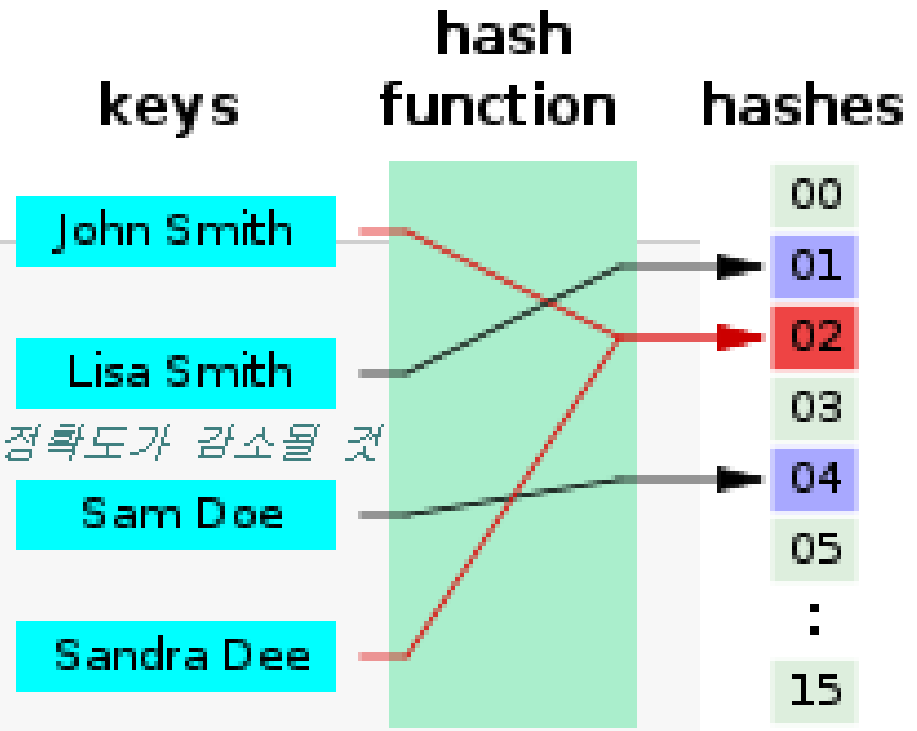
The cat sat on the mat.
1 0 0 0 0 0
0 1 0 0 0 0
0 0 1 0 0 0
...

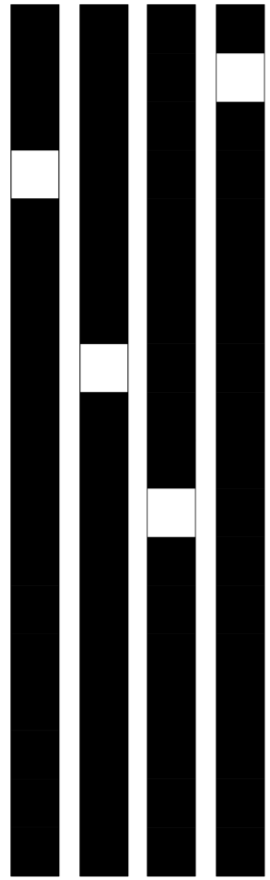
Found 9 unique tokens.

(9 tokens: The, cat, sat, on, mat, dog, ate, my, homework)

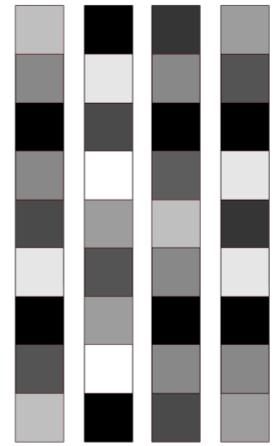
해싱 기법을 사용한 단어 수준의 원-핫 인코딩(간단한 예):

```
samples = ['The cat sat on the mat.', 'The dog ate my homework.']

# 단어를 크기가 1,000인 벡터로 저장합니다.
# 1,000개(또는 그이상)의 단어가 있다면 해싱 충돌이 늘어나고 인코딩의 정확도가 감소될 것
dimensionality = 1000
max_length = 10

results = np.zeros((len(samples), max_length, dimensionality))
for i, sample in enumerate(samples):
    for j, word in list(enumerate(sample.split()))[:max_length]:
        # 단어를 해싱하여 0과 1,000 사이의 랜덤한 정수 인덱스로 변환합니다.
        index = abs(hash(word)) % dimensionality
        results[i, j, index] = 1.
```





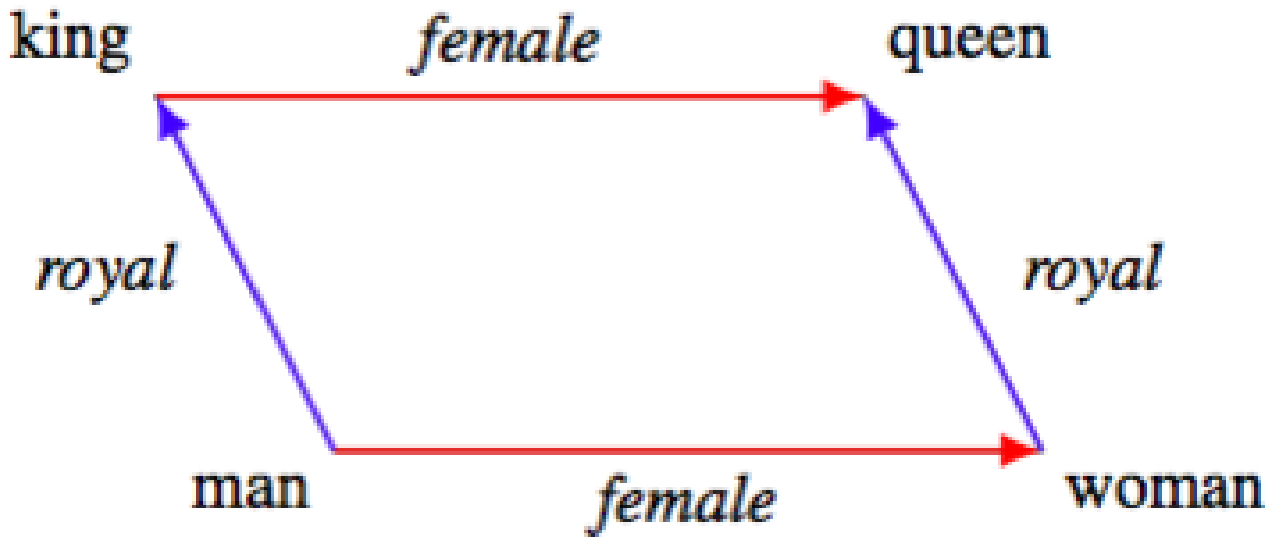
One-hot word vectors:
- Sparse
- High-dimensional
- Hard-coded



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

<Embedding 층을 사용하여 단어 임베딩 학습하기>

단어 임베딩은 언어를 기하학적 공간에 매핑하는 것이다.
예를들어 잘 구축된 임베딩 공간에서는 동의어가 비슷한 단어 벡터로 임베딩 된다.
일반적으로 두 단어 벡터 사이의 거리는 이 단어 사이의 의미 거리와 관계되어 있다.



```python
from keras.layers import Embedding

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

`Embedding` 층을 (특정 단어를 나타내는) 정수 인덱스를 밀집 벡터로 매핑하는 딕셔너리로 이해하는 것이 가장 좋습니다. 정수를 입력으로 받아 내부 딕셔너리에서 이 정수에 연관된 벡터를 찾아 반환합니다. 딕셔너리 탐색은 효율적으로 수행됩니다.

Embedding 층은 크기가 (samples, sequence_length)인 2D 정수 텐서를 입력으로 받는다.
예를 들어 Embedding 층에 (32, 10) 크기의 배치(길이가 10인 시퀀스 32개로 이루어진 배치)를 주입할 수 있다.

배치에 있는 모든 시퀀스 길이는 같아야 하므로 작은 길이의 시퀀스는 0으로 패딩되고 길이가 더 긴 시퀀스는 잘린다.

Embedding 층은 크기가 (samples, sequence_length, embedding_dimensionality)인 3D 실수형 텐서를 반환한다.

Embedding 층의 객체를 생성할 때 가중치는 다른 층과 마찬가지로 랜덤하게 초기화 된다.
훈련하면서 이 단어 벡터는 역전파를 통해 점차 조정된다.

IMDB 영화 리뷰 감성 예측 문제에 적용해보자.

```
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)

# 리스트를 (samples, maxlen) 크기의 2D 정수 텐서로 변환합니다.
x_train = preprocessing.sequence.pad_sequences(x_train, maxlen=maxlen)
x_test = preprocessing.sequence.pad_sequences(x_test, maxlen=maxlen)
```

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

model = Sequential()
# 나중에 임베딩된 입력을 Flatten 층에서 펼치기 위해 Embedding 층에 input_length를 지정합니다.
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())

# 분류기를 추가합니다.
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='rmsprop', loss='binary_crossentropy', metrics=['acc'])
model.summary()

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

Flatten으로 피는 이유는 위에 3차원으로 받았던 것을 아래 Sigmoid 함수를 사용해서 Dense층을 거쳐 하나의 숫자로 표현해야 하기 때문에 펴는 것이다.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_2 (Embedding)      (None, 20, 8)             80000
_________________________________________________________________
flatten_1 (Flatten)          (None, 160)               0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 161
=================================================================
Total params: 80,161
Trainable params: 80,161
Non-trainable params: 0
_________________________________________________________________
Train on 20000 samples, validate on 5000 samples
Epoch 1/10
20000/20000 [==============================] - 1s 74us/step - loss: 0.6759 - acc: 0.6050 - val_loss: 0.6398 - val_acc: 0.6814
Epoch 2/10
20000/20000 [==============================] - 1s 46us/step - loss: 0.5657 - acc: 0.7427 - val_loss: 0.5467 - val_acc: 0.7206
Epoch 3/10
20000/20000 [==============================] - 1s 46us/step - loss: 0.4752 - acc: 0.7808 - val_loss: 0.5113 - val_acc: 0.7384
Epoch 4/10
20000/20000 [==============================] - 1s 45us/step - loss: 0.4263 - acc: 0.8077 - val_loss: 0.5008 - val_acc: 0.7452
Epoch 5/10
20000/20000 [==============================] - 1s 45us/step - loss: 0.3930 - acc: 0.8258 - val_loss: 0.4981 - val_acc: 0.7538
Epoch 6/10
20000/20000 [==============================] - 1s 45us/step - loss: 0.3668 - acc: 0.8395 - val_loss: 0.5014 - val_acc: 0.7530
Epoch 7/10
20000/20000 [==============================] - 1s 45us/step - loss: 0.3435 - acc: 0.8533 - val_loss: 0.5052 - val_acc: 0.7520
Epoch 8/10
20000/20000 [==============================] - 1s 45us/step - loss: 0.3223 - acc: 0.8657 - val_loss: 0.5132 - val_acc: 0.7486
Epoch 9/10
20000/20000 [==============================] - 1s 46us/step - loss: 0.3022 - acc: 0.8766 - val_loss: 0.5213 - val_acc: 0.7490
Epoch 10/10
20000/20000 [==============================] - 1s 45us/step - loss: 0.2839 - acc: 0.8860 - val_loss: 0.5303 - val_acc: 0.7466
```

<사전 훈련된 임베딩 사용하기>

훈련 데이터가 부족하면 작업에 맞는 단어 임베딩을 학습할 수 없다. 이 때는 어떻게 해야 할까?

다른 문제에서 학습한 특성을 재사용하는 것이 합리적이다.

자연어 처리에서 사전 훈련된 임베딩을 사용하는 때는 충분한 데이터가 없어서 자신만의 좋은 특성을 학습하지 못하지만 꽤 일반적인 특성이 필요한 때이다

< 모든 내용을 적용하기: 원본 텍스트에서 단어 임베딩까지 >

```
import os

imdb_dir = './datasets/aclImdb'
train_dir = os.path.join(imdb_dir, 'train')

labels = []
texts = []

for label_type in ['neg', 'pos']:
    dir_name = os.path.join(train_dir, label_type)
    for fname in os.listdir(dir_name):
        if fname[-4:] == '.txt':
            f = open(os.path.join(dir_name, fname), encoding='utf8')
            texts.append(f.read())
            f.close()
            if label_type == 'neg':
                labels.append(0)
            else:
                labels.append(1)
```

```
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences
import numpy as np

maxlen = 100  # 100개 단어 이후는 버립니다
training_samples = 200  # 훈련 샘플은 200개입니다
validation_samples = 10000  # 검증 샘플은 10,000개입니다
max_words = 10000  # 데이터셋에서 가장 빈도 높은 10,000개의 단어만 사용합니다

tokenizer = Tokenizer(num_words=max_words)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)

word_index = tokenizer.word_index
print('%s개의 고유한 토큰을 찾았습니다.' % len(word_index))

data = pad_sequences(sequences, maxlen=maxlen)

labels = np.asarray(labels)
print('데이터 텐서의 크기:', data.shape)
print('레이블 텐서의 크기:', labels.shape)

# 데이터를 훈련 세트와 검증 세트로 분할합니다.
# 샘플이 순서대로 있기 때문에 (부정 샘플이 모두 나온 후에 긍정 샘플이 옵니다)
# 먼저 데이터를 섞습니다.
indices = np.arange(data.shape[0])
np.random.shuffle(indices)
data = data[indices]
labels = labels[indices]

x_train = data[:training_samples]
y_train = labels[:training_samples]
x_val = data[training_samples: training_samples + validation_samples]
y_val = labels[training_samples: training_samples + validation_samples]
```

```
88582개의 고유한 토큰을 찾았습니다.
데이터 텐서의 크기: (25000, 100)
레이블 텐서의 크기: (25000,)
```

임베딩 전처리

압축 해제한 파일(.txt 파일)을 파싱하여 단어(즉 문자열)와 이에 상응하는 벡터 표현(즉 숫자 벡터)를 매핑하는 인덱스를 만듭니다.

```
glove_dir = './datasets/'

embeddings_index = {}
f = open(os.path.join(glove_dir, 'glove.6B.100d.txt'), encoding="utf8")
for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
f.close()

print('%s개의 단어 벡터를 찾았습니다.' % len(embeddings_index))
```

예) { 민선 : [0.9, 6.8, 8.8, ...],
.....}

400000개의 단어 벡터를 찾았습니다.

	0	1	2	3	4	5
fox	-0.348680	-0.077720	0.177750	-0.094953	-0.452890	0.237790
ham	-0.773320	-0.282540	0.580760	0.841480	0.258540	0.585210
brown	-0.374120	-0.076264	0.109260	0.186620	0.029943	0.182700
beautiful	0.171200	0.534390	-0.348540	-0.097234	0.101800	-0.170860
jumps	-0.334840	0.215990	-0.350440	-0.260020	0.411070	0.154010
eggs	-0.417810	-0.035192	-0.126150	-0.215930	-0.669740	0.513250

```
embedding_dim = 100

      단어  순서
embedding_matrix = np.zeros((max_words, embedding_dim))
for word, i in word_index.items():
    embedding_vector = embeddings_index.get(word)
    if i < max_words:
        if embedding_vector is not None:
            # 임베딩 인덱스에 없는 단어는 모두 0이 됩니다.
            embedding_matrix[i] = embedding_vector
```

임베딩 백터를 매핑함

모델 정의하기

이전과 동일한 구조의 모델을 사용하겠습니다:

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

model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=maxlen))
model.add(Flatten())
model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_3 (Embedding)      (None, 100, 100)          1000000
_________________________________________________________________
flatten_2 (Flatten)          (None, 10000)             0
_________________________________________________________________
dense_2 (Dense)              (None, 32)                320032
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 33
=================================================================
Total params: 1,320,065
Trainable params: 1,320,065
Non-trainable params: 0
_________________________________________________________________
```

Param #은 계산되는 백터들이 이만큼 있다는 뜻

모델에 GloVe 임베딩 로드하기

`Embedding` 층은 하나의 가중치 행렬을 가집니다. 이 행렬은 2D 부동 소수 행렬이고 각 `i`번째 원소는 `i`번째 인덱스에 상응하는 단어 벡터입니다. 간단하네요. 모델의 첫 번째 층인 `Embedding` 층에 준비된 GloVe 행렬을 로드하세요:

```
model.layers[0].set_weights([embedding_matrix])
model.layers[0].trainable = False
```

추가적으로 `Embedding` 층을 동결합니다(`trainable` 속성을 `False`로 설정합니다). 사전 훈련된 컨브넷 특성을 사용할 때와 같은 이유입니다. 모델의 일부는 (`Embedding` 층처럼) 사전 훈련되고 다른 부분은 (최상단 분류기처럼) 랜덤하게 초기화되었다면 훈련하는 동안 사전 훈련된 부분이 업데이트되면 안됩니다. 이미 알고 있던 정보를 모두 잃게 됩니다. 랜덤하게 초기화된 층에서 대량의 그래디언트 업데이트가 발생하면 이미 학습된 특성을 오염시키기 때문입니다.

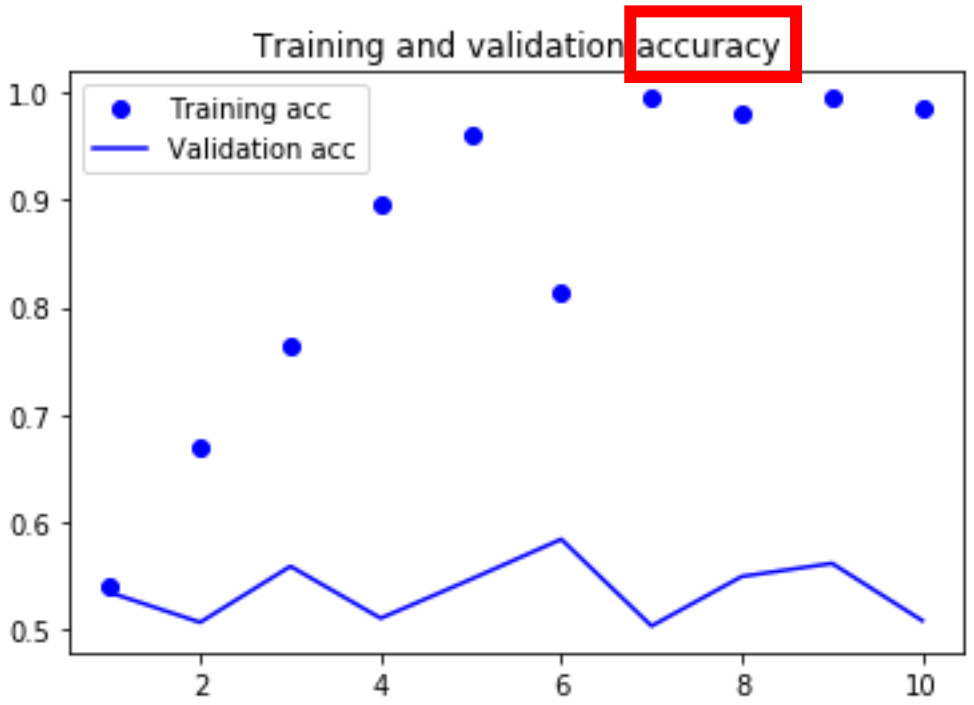
모델 훈련과 평가

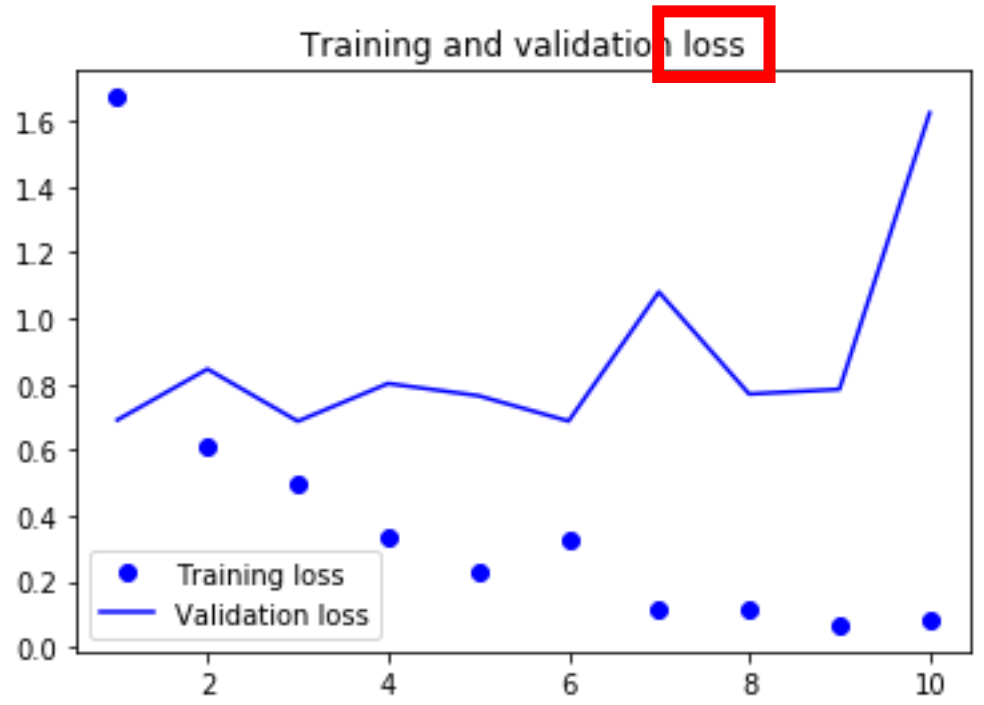
모델을 컴파일하고 훈련합니다:

```
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])
history = model.fit(x_train, y_train,
                    epochs=10,
                    batch_size=32,
                    validation_data=(x_val, y_val))
model.save_weights('pre_trained_glove_model.h5')
```

```
Train on 200 samples, validate on 10000 samples
Epoch 1/10
200/200 [==============================] - 0s 2ms/step - loss: 1.6733 - acc: 0.5400 - val_loss: 0.6907 - val_acc: 0.5338
Epoch 2/10
200/200 [==============================] - 0s 974us/ste - loss: 0.6077 - acc: 0.6700 - val_loss: 0.8459 - val_acc: 0.5060
Epoch 3/10
200/200 [==============================] - 0s 988us/ste - loss: 0.4950 - acc: 0.7650 - val_loss: 0.6870 - val_acc: 0.5583
Epoch 4/10
200/200 [==============================] - 0s 1ms/step - loss: 0.3349 - acc: 0.8950 - val_loss: 0.8019 - val_acc: 0.5098
Epoch 5/10
200/200 [==============================] - 0s 966us/ste - loss: 0.2251 - acc: 0.9600 - val_loss: 0.7647 - val_acc: 0.5462
Epoch 6/10
200/200 [==============================] - 0s 1ms/step - loss: 0.3254 - acc: 0.8150 - val_loss: 0.6875 - val_acc: 0.5835
Epoch 7/10
200/200 [==============================] - 0s 1ms/step - loss: 0.1144 - acc: 0.9950 - val_loss: 1.0806 - val_acc: 0.5026
Epoch 8/10
200/200 [==============================] - 0s 1ms/step - loss: 0.1141 - acc: 0.9800 - val_loss: 0.7699 - val_acc: 0.5488
Epoch 9/10
200/200 [==============================] - 0s 982us/ste - loss: 0.0671 - acc: 0.9950 - val_loss: 0.7842 - val_acc: 0.5610
Epoch 10/10
200/200 [==============================] - 0s 1ms/step - loss: 0.0814 - acc: 0.9850 - val_loss: 1.6252 - val_acc: 0.5076
```

훈련 샘플 수가 적기 때문에 과대적합이 빠르게 시작됨.





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

model = Sequential()
model.add(Embedding(max_words, embedding_dim, input_length=maxlen))
model.add(Flatten())
model.add(Dense(32, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.summary()

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])
history = model.fit(x_train, y_train,
                    epochs=10,
                    batch_size=32,
                    validation_data=(x_val, y_val))
```

Layer (type)	Output Shape	Param #
embedding_4 (Embedding)	(None, 100, 100)	1000000
flatten_3 (Flatten)	(None, 10000)	0
dense_4 (Dense)	(None, 32)	320032
dense_5 (Dense)	(None, 1)	33

Total params: 1,320,065
Trainable params: 1,320,065
Non-trainable params: 0

```
Train on 200 samples, validate on 10000 samples
Epoch 1/10
200/200 [==============================] - 1s 3ms/step - loss: 0.6894 - acc: 0.5200 - val_loss: 0.6953 - val_acc: 0.5117
Epoch 2/10
200/200 [==============================] - 0s 1ms/step - loss: 0.4895 - acc: 0.9650 - val_loss: 0.7149 - val_acc: 0.5138
Epoch 3/10
200/200 [==============================] - 0s 1ms/step - loss: 0.2904 - acc: 0.9800 - val_loss: 0.7029 - val_acc: 0.5150
Epoch 4/10
200/200 [==============================] - 0s 1ms/step - loss: 0.1236 - acc: 1.0000 - val_loss: 0.7255 - val_acc: 0.5194
Epoch 5/10
200/200 [==============================] - 0s 1ms/step - loss: 0.0553 - acc: 1.0000 - val_loss: 0.7111 - val_acc: 0.5195
Epoch 6/10
200/200 [==============================] - 0s 1ms/step - loss: 0.0275 - acc: 1.0000 - val_loss: 0.7580 - val_acc: 0.5212
Epoch 7/10
200/200 [==============================] - 0s 1ms/step - loss: 0.0152 - acc: 1.0000 - val_loss: 0.7334 - val_acc: 0.5197
Epoch 8/10
200/200 [==============================] - 0s 1ms/step - loss: 0.0085 - acc: 1.0000 - val_loss: 0.7509 - val_acc: 0.5228
Epoch 9/10
200/200 [==============================] - 0s 1ms/step - loss: 0.0052 - acc: 1.0000 - val_loss: 0.7437 - val_acc: 0.5208
Epoch 10/10
200/200 [==============================] - 0s 1ms/step - loss: 0.0032 - acc: 1.0000 - val_loss: 0.7570 - val_acc: 0.5243
```

loss: 0.6894 - acc: 0.5200 - val_loss: 0.6953 - val_acc: 0.5117

loss: 0.4895 - acc: 0.9650 - val_loss: 0.7149 - val_acc: 0.5138

loss: 0.2904 - acc: 0.9800 - val_loss: 0.7029 - val_acc: 0.5150

loss: 0.1236 - acc: 1.0000 - val_loss: 0.7255 - val_acc: 0.5194

loss: 0.0553 - acc: 1.0000 - val_loss: 0.7111 - val_acc: 0.5195

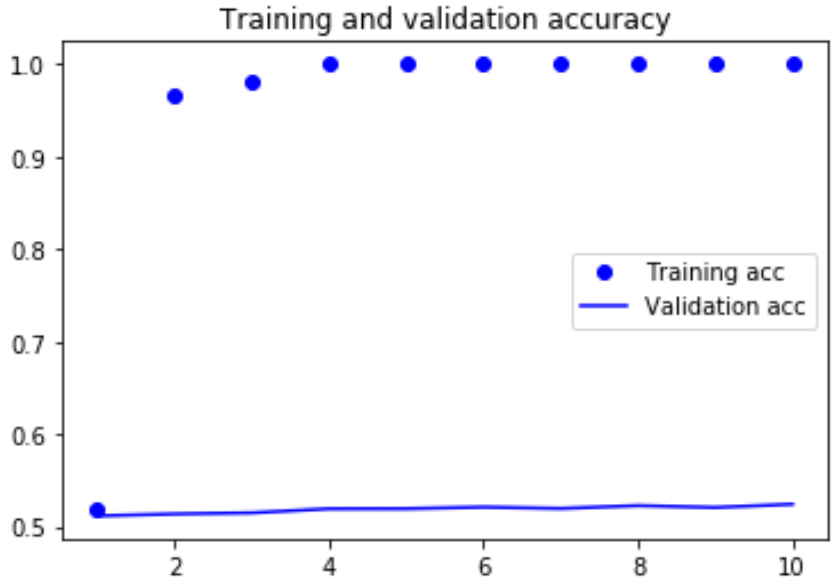
loss: 0.0275 - acc: 1.0000 - val_loss: 0.7580 - val_acc: 0.5212

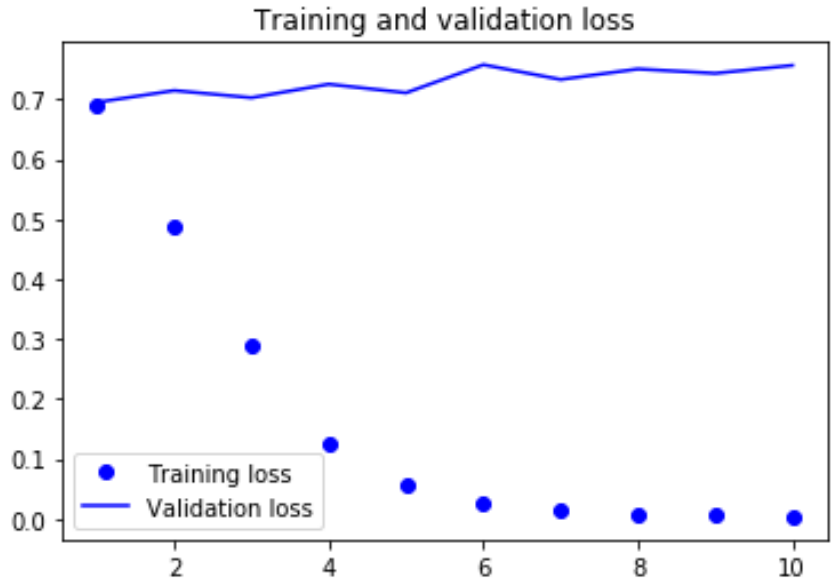
loss: 0.0152 - acc: 1.0000 - val_loss: 0.7334 - val_acc: 0.5197

loss: 0.0085 - acc: 1.0000 - val_loss: 0.7509 - val_acc: 0.5228

loss: 0.0052 - acc: 1.0000 - val_loss: 0.7437 - val_acc: 0.5208

loss: 0.0032 - acc: 1.0000 - val_loss: 0.7570 - val_acc: 0.5243





훈련된 데이터를 가져오지 않고 동결 없이 진행한 결과입니다.

앞에 훈련된 데이터를 사용한 것보다 더 결과가 좋지 않습니다.

<훈련 데이터를 2,000개로 늘렸을 때의 결과>

```
Train on 2000 samples, validate on 10000 samples
Epoch 1/10
2000/2000 [==============================] - 0s 174us/step - loss: 0.6383 - acc: 0.6060 - val_loss: 0.6543 - val_acc: 0.6142
Epoch 2/10
2000/2000 [==============================] - 0s 174us/step - loss: 0.1578 - acc: 0.9880 - val_loss: 0.6246 - val_acc: 0.6631
Epoch 3/10
2000/2000 [==============================] - 0s 174us/step - loss: 0.0191 - acc: 0.9995 - val_loss: 0.6568 - val_acc: 0.6787
Epoch 4/10
2000/2000 [==============================] - 0s 175us/step - loss: 0.0016 - acc: 1.0000 - val_loss: 0.7071 - val_acc: 0.6916
Epoch 5/10
2000/2000 [==============================] - 0s 171us/step - loss: 1.2005e-04 - acc: 1.0000 - val_loss: 0.7635 - val_acc: 0.6972
Epoch 6/10
2000/2000 [==============================] - 0s 173us/step - loss: 8.1681e-06 - acc: 1.0000 - val_loss: 0.8398 - val_acc: 0.7043
Epoch 7/10
2000/2000 [==============================] - 0s 170us/step - loss: 7.4167e-07 - acc: 1.0000 - val_loss: 0.9032 - val_acc: 0.7046
Epoch 8/10
2000/2000 [==============================] - 0s 172us/step - loss: 1.7394e-07 - acc: 1.0000 - val_loss: 0.9659 - val_acc: 0.7041
Epoch 9/10
2000/2000 [==============================] - 0s 172us/step - loss: 1.1575e-07 - acc: 1.0000 - val_loss: 0.9996 - val_acc: 0.7038
Epoch 10/10
2000/2000 [==============================] - 0s 170us/step - loss: 1.1111e-07 - acc: 1.0000 - val_loss: 1.0031 - val_acc: 0.7046
```

< 이제까지는 훈련데이터였고, 여기부터 테스트 데이터 사용 >

```
test_dir = os.path.join(imdb_dir, 'test')

labels = []
texts = []

for label_type in ['neg', 'pos']:
    dir_name = os.path.join(test_dir, label_type)
    for fname in sorted(os.listdir(dir_name)):
        if fname[-4:] == '.txt':
            f = open(os.path.join(dir_name, fname), encoding="utf8")
            texts.append(f.read())
            f.close()
            if label_type == 'neg':
                labels.append(0)
            else:
                labels.append(1)

sequences = tokenizer.texts_to_sequences(texts)
x_test = pad_sequences(sequences, maxlen=maxlen)
y_test = np.asarray(labels)
```

그다음 이 절의 첫 번째 모델을 로드하고 평가합니다:

```
model.load_weights('pre_trained_glove_model.h5')
model.evaluate(x_test, y_test)
```

```
25000/25000 [==============================] - 1s 21us/step

[1.6611276818060874, 0.50412]
```

테스트 정확도는 겨우 50% 정도입니다. 적은 수의 훈련 샘플로 작업하는 것은 어려운 일이군요!

이제 다음 작업을 할 수 있다.

- 원본 텍스트를 신경망이 처리할 수 있는 형태로 변환

- 케라스 모델에 Embedding 층을 추가하여 어떤 작업에 특화된 토큰 임베딩을 학습

- 데이터가 부족한 자연어 처리 문제에서 사전 훈련된 단어 임베딩을 사용하여 성능 향상을 꾀함