

사전 훈련된 단어 임베딩 사용

- 데이터가 적을 경우, 훈련 데이터 부족으로 인해 작업에 맞는 단어 임베딩이 어려울 수 있다.
- 이때 사전 훈련 알고리즘이 힘을 발휘할 수 있다.
- 단어 임베딩은 일반적으로 (문장이나 문서에 같이 등장하는 단어 관찰) 단어 출현 통계를 사용하여 계산.
- 케라스의 Embedding 층을 사용하기 위해 다운로드가 가능한 미리 계산된 단어 임베딩 데이터베이스가 많이 있다.

유명한 알고리즘

- 구글의 토마스 미코로프의 word2vec 알고리즘(<https://code.google.com/archive/p/word2vec>)
(<https://code.google.com/archive/p/word2vec>)
 - 2013년 구글의 토마스 미코로프가 개발.
- 스탠포드 대학교 : GloVe(<https://nlp.stanford.edu/projects/glove>) (<https://nlp.stanford.edu/projects/glove>) -
2014년 스탠포드 대학교 연구자들
 - 위키피디아 데이터와 커먼 크롤 데이터에서 가져온 수백만 개의 영어 토큰에 대해 임베딩을 계산해 둠.

In [5]:



```
from IPython.display import display, Image
import os, warnings

warnings.filterwarnings(action='ignore')
```

01 데이터 준비

- 데이터 다운로드 URL : <http://mng.bz/0tlo> (<http://mng.bz/0tlo>) (케라스 책 참조)
- 현재 ipynb 노트북의 상위 폴더(data_lmdb)에 압축 파일을 풀어준다.

In [6]:



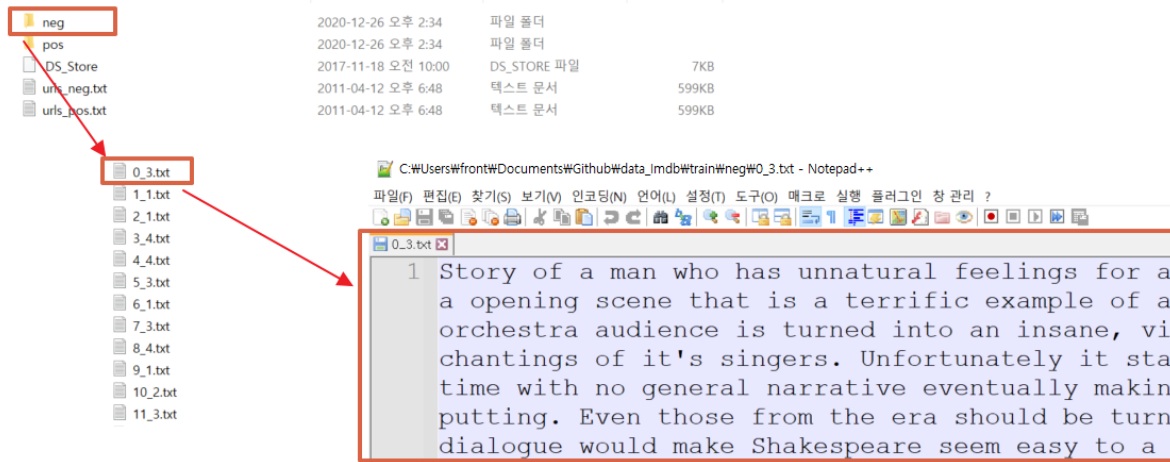
```
import os

imdb_dir = '../data_lmdb'
train_dir = os.path.join(imdb_dir, 'train')
```

- 데이터 폴더의 neg, pos 폴더의 파일 하나씩을 불러와. 이를 texts에 추가.
- labels에는 긍정인지, 아닌지를 추가

In [7]:

```
display(Image(filename="img/imdb_data.png"))
```



In [9]:

```
%%time

labels = []
texts = []

# 01. neg, pos 각각의 폴더의 파일들을 확인
# 02. txt 파일을 확인 후, 파일 내용을 확인하고 texts에 추가
# 03. 폴더명이 'neg'이면 (0)을 labels에 추가, 'pos'이면 (1)을 labels에 추가
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)
```

Wall time: 41.6 s

02. 학습용 데이터 1000개의 샘플로 영화 리뷰 분류

- 이전에서 학습한 것을 이용하여 텍스트를 벡터로 만들고, 훈련 세트와 검증 세트로 나눈다.
- 사전 훈련된 단어 임베딩은 훈련 데이터가 부족한 문제에 특히 유용

In [11]:

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

In [12]:

```
max_words = 10000    # 데이터셋에서 가장 빈도 높은 10,000개의 단어만 사용.
maxlen = 100         # 100개 단어까지 가져오기(뒤에서부터)

tr_samples = 1000    # 훈련 샘플은 1000개입니다
val_samples = 10000  # 검증 샘플은 10,000개입니다
```

케라스의 Tokenizer를 이용하여 텍스트를 '단어:정수'로 변환

In [13]:

```
tokenizer = Tokenizer(num_words=max_words)
tokenizer.fit_on_texts(texts) # 텍스트 목록을 기반으로 어휘 업데이트
```

단어와 정수의 딕셔너리 형태 확인

In [14]:

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

73998개의 고유한 토큰을 찾았습니다.

```
{'the': 1, 'a': 2, 'and': 3, 'of': 4, 'to': 5, 'is': 6, 'br': 7, 'in': 8, 'i': 9,
 'it': 10, 'this': 11, 'that': 12, 'was': 13, 'movie': 14, 'as': 15, 'for': 16, 'w
 ith': 17, 'but': 18, 'film': 19, 'on': 20, 'not': 21, 'you': 22, 'are': 23, 'hav
 e': 24, 'be': 25, 'his': 26, 'one': 27, 'he': 28, 'all': 29, 'at': 30, 'they': 3
 1, 'by': 32, 'so': 33, 'like': 34, 'an': 35, 'from': 36, 'who': 37, 'just': 38,
 'or': 39, 'her': 40, 'if': 41, 'about': 42, 'out': 43, "it's": 44, 'there': 45,
 'has': 46, 'some': 47, 'what': 48, 'good': 49, 'no': 50, 'more': 51, 'when': 52,
 'even': 53, 'up': 54, 'very': 55, 'would': 56, 'only': 57, 'she': 58, 'time': 59,
 'my': 60, 'really': 61, 'had': 62, 'which': 63, 'bad': 64, 'story': 65, 'were': 6
 6, 'see': 67, 'their': 68, 'can': 69, 'me': 70, 'than': 71, 'much': 72, 'well': 7
 3, 'been': 74, 'we': 75, 'do': 76, 'get': 77, 'because': 78, "don't": 79, 'peopl
 e': 80, 'how': 81, 'into': 82, 'other': 83, 'first': 84, 'made': 85, 'then': 86,
 'will': 87, 'make': 88, 'most': 89, 'any': 90, 'could': 91, 'also': 92, 'him': 9
 3, 'too': 94, 'them': 95, 'way': 96, 'movies': 97, 'after': 98, 'its': 99, 'grea
 t': 100, 'think': 101, 'acting': 102, 'plot': 103, 'characters': 104, 'watch': 10
 5, 'character': 106, 'being': 107, 'two': 108, 'seen': 109, 'never': 110, 'did':
 111, 'films': 112, 'off': 113, 'show': 114, 'over': 115, 'know': 116, 'little': 1
 17, 'where': 118, 'ever': 119, 'many': 120, 'better': 121, 'why': 122, 'your': 12
```

In [21]:



```
sequences = tokenizer.texts_to_sequences(texts) # 텍스트로 정수로 변환
print(type(sequences) )
print(sequences[0:1])
```

```
<class 'list'>
[[65, 4, 2, 142, 37, 46, 6155, 1520, 16, 2, 3717, 483, 43, 17, 2, 631, 130, 12, 6,
2, 1715, 437, 4, 1504, 210, 2, 298, 6, 636, 82, 35, 2094, 1167, 2957, 32, 1, 884, 4,
44, 5441, 410, 10, 2995, 1504, 1, 209, 59, 17, 50, 808, 1419, 911, 215, 10, 38, 94,
113, 1512, 53, 152, 36, 1, 1105, 136, 25, 636, 113, 1, 385, 56, 88, 2136, 299, 908,
5, 2, 837, 9573, 20, 2, 1846, 657, 44, 121, 71, 22, 222, 101, 17, 47, 49, 693, 32, 8
59, 100, 859, 381, 2996, 3, 9321, 69, 25, 109, 3227]]
```

In [22]:



```
np.ndim(sequences), len(sequences), type(sequences)
```

Out[22]:

```
(1, 17597, list)
```

In [23]:



```
data = pad_sequences(sequences, maxlen=maxlen)

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

```
데이터 텐서의 크기: (17597, 100)
레이블 텐서의 크기: (17597,)
```

In [42]:



```
# tr_samples = 200    # 훈련 샘플은 200개입니다
tr_samples = 200      # 훈련 샘플은 200개입니다
val_samples = 10000    # 검증 샘플은 10,000개입니다
```

데이터를 학습용 세트와 검증 세트로 분할

In [43]:



```
# 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[:tr_samples]
y_train = labels[:tr_samples]
X_val = data[tr_samples: tr_samples + val_samples]
y_val = labels[tr_samples: tr_samples + val_samples]

X_train.shape, X_val.shape, y_train.shape, y_val.shape
```

Out[43]:

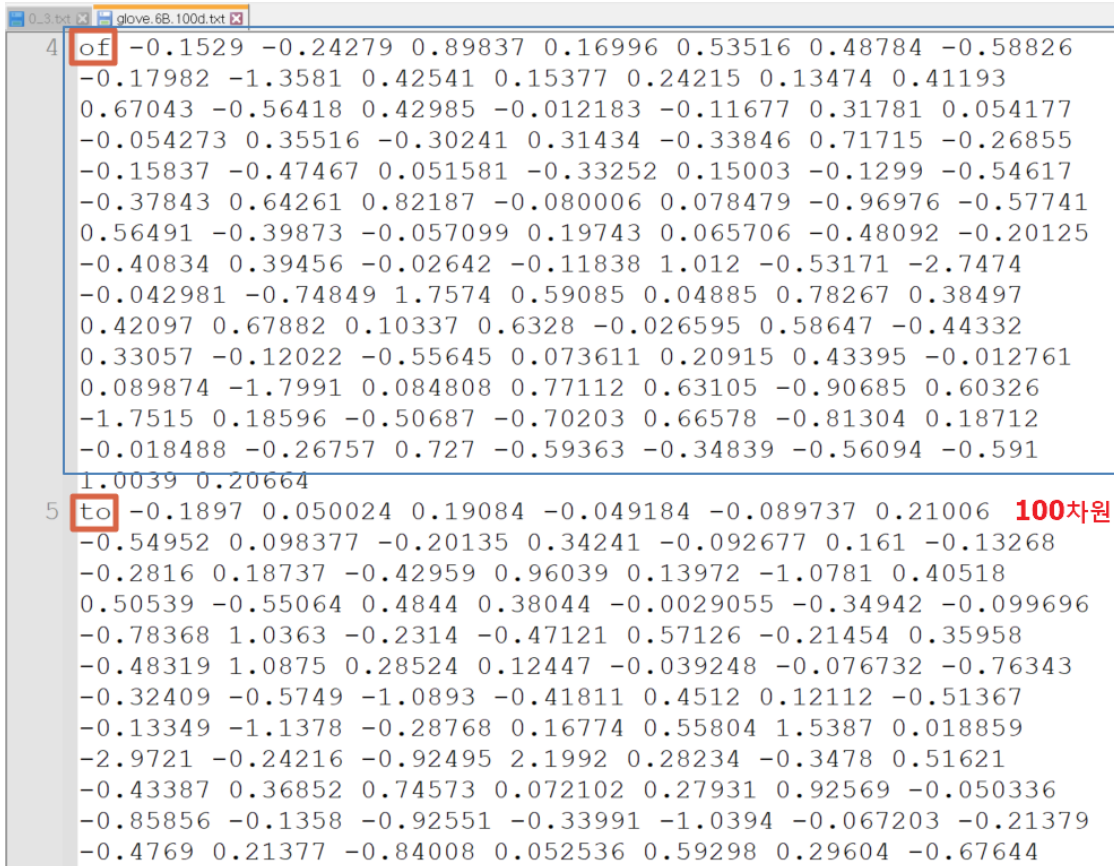
```
((200, 100), (10000, 100), (200,), (10000,))
```

Glove 단어 임베딩

- 내려받기 : <https://nlp.stanford.edu/projects/glove> (<https://nlp.stanford.edu/projects/glove>)
 - Download pre-trained word vectors
 - Wikipedia 2014 + Gigaword 5 (6B tokens, 400K vocab, uncased, 50d, 100d, 200d, & 300d vectors, 822 MB download): glove.6B.zip
 - 40만개 단어(또는 단어가 아닌 토큰)에 대한 100차원의 임베딩 벡터를 포함

In [44]:

```
display(Image(filename="img/glove_6B_100d.png"))
```



```

4 of -0.1529 -0.24279 0.89837 0.16996 0.53516 0.48784 -0.58826
-0.17982 -1.3581 0.42541 0.15377 0.24215 0.13474 0.41193
0.67043 -0.56418 0.42985 -0.012183 -0.11677 0.31781 0.054177
-0.054273 0.35516 -0.30241 0.31434 -0.33846 0.71715 -0.26855
-0.15837 -0.47467 0.051581 -0.33252 0.15003 -0.1299 -0.54617
-0.37843 0.64261 0.82187 -0.080006 0.078479 -0.96976 -0.57741
0.56491 -0.39873 -0.057099 0.19743 0.065706 -0.48092 -0.20125
-0.40834 0.39456 -0.02642 -0.11838 1.012 -0.53171 -2.7474
-0.042981 -0.74849 1.7574 0.59085 0.04885 0.78267 0.38497
0.42097 0.67882 0.10337 0.6328 -0.026595 0.58647 -0.44332
0.33057 -0.12022 -0.55645 0.073611 0.20915 0.43395 -0.012761
0.089874 -1.7991 0.084808 0.77112 0.63105 -0.90685 0.60326
-1.7515 0.18596 -0.50687 -0.70203 0.66578 -0.81304 0.18712
-0.018488 -0.26757 0.727 -0.59363 -0.34839 -0.56094 -0.591
1.0039 0.20664
5 to -0.1897 0.050024 0.19084 -0.049184 -0.089737 0.21006 100차원
-0.54952 0.098377 -0.20135 0.34241 -0.092677 0.161 -0.13268
-0.2816 0.18737 -0.42959 0.96039 0.13972 -1.0781 0.40518
0.50539 -0.55064 0.4844 0.38044 -0.0029055 -0.34942 -0.099696
-0.78368 1.0363 -0.2314 -0.47121 0.57126 -0.21454 0.35958
-0.48319 1.0875 0.28524 0.12447 -0.039248 -0.076732 -0.76343
-0.32409 -0.5749 -1.0893 -0.41811 0.4512 0.12112 -0.51367
-0.13349 -1.1378 -0.28768 0.16774 0.55804 1.5387 0.018859
-2.9721 -0.24216 -0.92495 2.1992 0.28234 -0.3478 0.51621
-0.43387 0.36852 0.74573 0.072102 0.27931 0.92569 -0.050336
-0.85856 -0.1358 -0.92551 -0.33991 -1.0394 -0.067203 -0.21379
-0.4769 0.21377 -0.84008 0.052536 0.59298 0.29604 -0.67644

```

In [45]:

```

glove_dir = '../data_lmdb/glove'

embeddings_index = {}

# 파일 읽어오기
f = open(os.path.join(glove_dir, 'glove.6B.100d.txt'), encoding="utf8")

# 첫번째가 단어
# 나머지는 벡터값 수치
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))

```

400000개의 단어 벡터를 찾았습니다.

In [46]:



```
type( word_index.items() )
```

Out[46]:

```
dict_items
```

In [47]:



```
word_index.items()

cnt = 0
for word, i in word_index.items():
    print(word,i)
    cnt += 1
    if cnt==15:
        break
```

```
the 1
a 2
and 3
of 4
to 5
is 6
br 7
in 8
i 9
it 10
this 11
that 12
was 13
movie 14
as 15
```

In [48]:



```
embeddings_index.get('the')
```

Out[48]:

```
array([-0.038194, -0.24487 ,  0.72812 , -0.39961 ,  0.083172,  0.043953,
       -0.39141 ,  0.3344  , -0.57545 ,  0.087459,  0.28787 , -0.06731 ,
        0.30906 , -0.26384 , -0.13231 , -0.20757 ,  0.33395 , -0.33848 ,
       -0.31743 , -0.48336 ,  0.1464  , -0.37304 ,  0.34577 ,  0.052041,
        0.44946 , -0.46971 ,  0.02628 , -0.54155 , -0.15518 , -0.14107 ,
       -0.039722,  0.28277 ,  0.14393 ,  0.23464 , -0.31021 ,  0.086173,
        0.20397 ,  0.52624 ,  0.17164 , -0.082378, -0.71787 , -0.41531 ,
        0.20335 , -0.12763 ,  0.41367 ,  0.55187 ,  0.57908 , -0.33477 ,
       -0.36559 , -0.54857 , -0.062892,  0.26584 ,  0.30205 ,  0.99775 ,
       -0.80481 , -3.0243  ,  0.01254 , -0.36942 ,  2.2167  ,  0.72201 ,
       -0.24978 ,  0.92136 ,  0.034514,  0.46745 ,  1.1079  , -0.19358 ,
       -0.074575,  0.23353 , -0.052062, -0.22044 ,  0.057162, -0.15806 ,
       -0.30798 , -0.41625 ,  0.37972 ,  0.15006 , -0.53212 , -0.2055  ,
       -1.2526  ,  0.071624,  0.70565 ,  0.49744 , -0.42063 ,  0.26148 ,
       -1.538   , -0.30223 , -0.073438, -0.28312 ,  0.37104 , -0.25217 ,
        0.016215, -0.017099, -0.38984 ,  0.87424 , -0.72569 , -0.51058 ,
       -0.52028 , -0.1459  ,  0.8278  ,  0.27062 ], dtype=float32)
```

In [49]:



```
# max_words = 10000 # 데이터셋에서 가장 빈도 높은 10,000개의 단어만 사용.
```

In [50]:



```
# 01. embedding_matrix을 0으로 초기화
# 02. 각 단어별 100차원의 가중치를 사전 네트워크 가중치로 초기화
embedding_dim = 100
embedding_matrix = np.zeros((max_words, embedding_dim))
print(embedding_matrix.shape)
for word, i in word_index.items():
    embedding_vector = embeddings_index.get(word) # 각 단어별 가중치
    if i < max_words:
        if embedding_vector is not None:

            # 임베딩 인덱스에 없는 단어는 모두 0이 됩니다.
            embedding_matrix[i] = embedding_vector
```

```
(10000, 100)
```


In [51]:



```
embedding_matrix[0:2]
```

Out[51]:

```
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.          ,  0.          ,  0.          ,  0.          ,  0.          ,
         0.          ,  0.          ,  0.          ,  0.          ,  0.          ,
         0.          ,  0.          ,  0.          ,  0.          ,  0.          ,
         0.          ,  0.          ,  0.          ,  0.          ,  0.          ],
       [-0.038194 , -0.24487001,  0.72812003, -0.39961001,  0.083172 ,
         0.043953 , -0.39140999,  0.3344      , -0.57545  ,  0.087459 ,
         0.28786999, -0.06731  ,  0.30906001, -0.26383999, -0.13231  ,
        -0.20757  ,  0.33395001, -0.33848  , -0.31742999, -0.48335999,
         0.1464    , -0.37303999,  0.34577  ,  0.052041 ,  0.44946  ,
        -0.46970999,  0.02628  , -0.54154998, -0.15518001, -0.14106999,
        -0.039722 ,  0.28277001,  0.14393  ,  0.23464  , -0.31020999,
         0.086173 ,  0.20397  ,  0.52623999,  0.17163999, -0.082378 ,
        -0.71787  , -0.41531  ,  0.20334999, -0.12763  ,  0.41367  ,
         0.55186999,  0.57907999, -0.33476999, -0.36559001, -0.54856998,
        -0.062892 ,  0.26583999,  0.30204999,  0.99774998, -0.80480999,
        -3.0243001 ,  0.01254  , -0.36941999,  2.21670008,  0.72201002,
        -0.24978  ,  0.92136002,  0.034514  ,  0.46744999,  1.10790002,
        -0.19358  , -0.074575 ,  0.23353  , -0.052062 , -0.22044  ,
         0.057162 , -0.15806  , -0.30798  , -0.41624999,  0.37972  ,
         0.15006  , -0.53211999, -0.20550001, -1.25259995,  0.071624  ,
         0.70564997,  0.49744001, -0.42063001,  0.26148  , -1.53799999,
        -0.30223  , -0.073438 , -0.28312001,  0.37103999, -0.25217  ,
         0.016215 , -0.017099 , -0.38984001,  0.87423998, -0.72569001,
        -0.51058  , -0.52028  , -0.1459    ,  0.82779998,  0.27061999]])
```

모델 구현

In [52]:



```
max_words, embedding_dim, maxlen
```

Out[52]:

```
(10000, 100, 100)
```

모델 구축

In [53]:

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

# 10000개의 단어(샘플수)를 100개 차원 임베딩
# input_length : 시퀀스의 길이
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: "sequential_2"

Layer (type)	Output Shape	Param #
embedding_2 (Embedding)	(None, 100, 100)	1000000
flatten_2 (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		

모델에 GloVe 임베딩 로드하기

- Embedding 층은 하나의 가중치 행렬을 가집니다.
- 이 행렬은 2D 부동 소수 행렬이고 각 i번째 원소는 i번째 인덱스에 상응하는 단어 벡터입니다.
- 모델의 첫 번째 층인 Embedding 층에 준비된 GloVe 행렬을 로드

In [54]:

```
embedding_matrix.shape
```

Out [54]:

(10000, 100)

In [55]:

```
model.layers[0].set_weights([embedding_matrix])
model.layers[0].trainable = False # 추가적으로 Embedding을 동결
```

모델 훈련과 평가

모델의 성능을 그래프로 그려보겠다.

In [56]:



```
%%time

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')
```

```
Epoch 1/10
7/7 [=====] - 2s 149ms/step - loss: 1.7266 - acc: 0.5710 -
val_loss: 0.5977 - val_acc: 0.7057
Epoch 2/10
7/7 [=====] - 1s 108ms/step - loss: 0.6651 - acc: 0.7247 -
val_loss: 0.6093 - val_acc: 0.7015
Epoch 3/10
7/7 [=====] - 1s 114ms/step - loss: 0.4555 - acc: 0.8101 -
val_loss: 0.6870 - val_acc: 0.7063
Epoch 4/10
7/7 [=====] - 1s 112ms/step - loss: 0.2316 - acc: 0.9298 -
val_loss: 0.6673 - val_acc: 0.6280
Epoch 5/10
7/7 [=====] - 1s 107ms/step - loss: 0.1127 - acc: 0.9758 -
val_loss: 1.3681 - val_acc: 0.7057
Epoch 6/10
7/7 [=====] - 1s 108ms/step - loss: 0.0757 - acc: 0.9656 -
val_loss: 0.8051 - val_acc: 0.7055
Epoch 7/10
7/7 [=====] - 1s 111ms/step - loss: 0.0316 - acc: 1.0000 -
val_loss: 0.7738 - val_acc: 0.7057
Epoch 8/10
7/7 [=====] - 1s 108ms/step - loss: 0.0136 - acc: 1.0000 -
val_loss: 0.9139 - val_acc: 0.7062
Epoch 9/10
7/7 [=====] - 1s 112ms/step - loss: 0.0108 - acc: 0.9988 -
val_loss: 4.9403 - val_acc: 0.2949
Epoch 10/10
7/7 [=====] - 1s 103ms/step - loss: 1.2749 - acc: 0.6997 -
val_loss: 0.8584 - val_acc: 0.7031
Wall time: 7.62 s
```

성능 그래프 확인

In [57]:

```
import matplotlib.pyplot as plt

acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

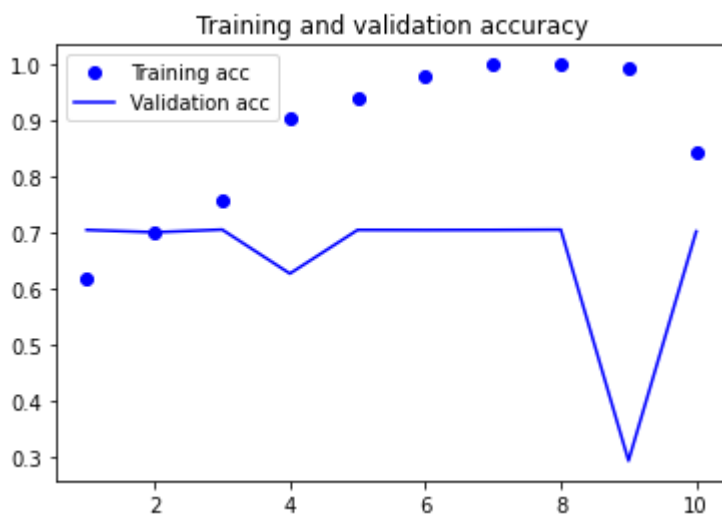
epochs = range(1, len(acc) + 1)

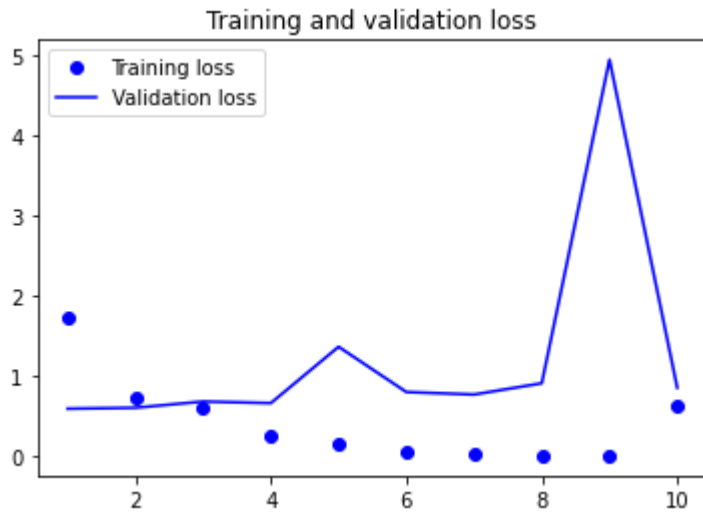
plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()

plt.figure()

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()
```





실습

- 학습용 데이터 셋을 200개에서 1000개로 올려보자.
 - 그렇다면 여기에서 추가적으로 데이터의 수량을 키우면 정확도를 올리면 어떻게 될까?

03 사전 훈련된 단어 임베딩을 사용하지 않고, 같은 모델 훈련해 보기

In []:



```

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))

model.save_weights('basic_trained_glove_model.h5')

```

Model: "sequential_3"

Layer (type)	Output Shape	Param #
embedding_3 (Embedding)	(None, 100, 100)	1000000
flatten_3 (Flatten)	(None, 10000)	0
dense_6 (Dense)	(None, 32)	320032
dense_7 (Dense)	(None, 1)	33

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

Epoch 1/10

7/7 [=====] - 1s 139ms/step - loss: 0.6769 - acc: 0.5842 - val_loss: 0.6277 - val_acc: 0.7057

Epoch 2/10

7/7 [=====] - 1s 108ms/step - loss: 0.4245 - acc: 0.8627 - val_loss: 0.6188 - val_acc: 0.7056

Epoch 3/10

7/7 [=====] - 1s 114ms/step - loss: 0.2320 - acc: 0.9927 - val_loss: 0.6262 - val_acc: 0.7035

Epoch 4/10

7/7 [=====] - 1s 112ms/step - loss: 0.0982 - acc: 1.0000 - val_loss: 0.6464 - val_acc: 0.7056

Epoch 5/10

7/7 [=====] - 1s 111ms/step - loss: 0.0474 - acc: 1.0000 - val_loss: 0.6568 - val_acc: 0.7054

Epoch 6/10

7/7 [=====] - 1s 103ms/step - loss: 0.0246 - acc: 1.0000 - val_loss: 0.6537 - val_acc: 0.7047

Epoch 7/10

7/7 [=====] - 1s 111ms/step - loss: 0.0135 - acc: 1.0000 - val_loss: 0.6768 - val_acc: 0.7047

Epoch 8/10

7/7 [=====] - 1s 112ms/step - loss: 0.0075 - acc: 1.0000 -

val_loss: 0.6904 - val_acc: 0.7048

Epoch 9/10

6/7 [=====>.....] - ETA: 0s - loss: 0.0047 - acc: 1.0000

In [52]:

```
acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

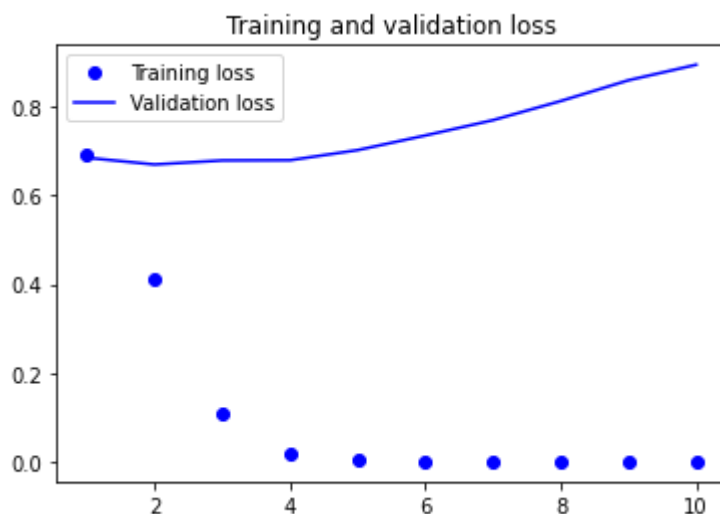
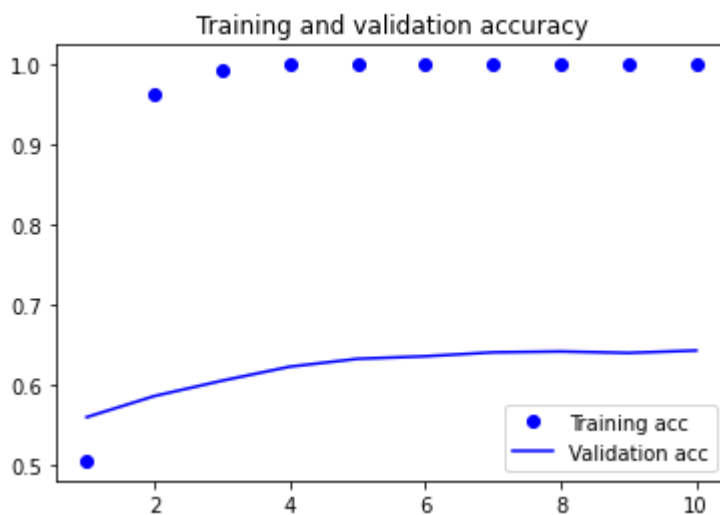
epochs = range(1, len(acc) + 1)

plt.plot(epochs, acc, 'bo', label='Training acc')
plt.plot(epochs, val_acc, 'b', label='Validation acc')
plt.title('Training and validation accuracy')
plt.legend()

plt.figure()

plt.plot(epochs, loss, 'bo', label='Training loss')
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.legend()

plt.show()
```



훈련 모델이 더 낫다. 64%

- 실습해 보기. 데이터를 더 늘리거나, 줄이면 어떻게 되는가?

04 테스트 데이터에서 모델을 평가해보기

In [54]:

```
%%time

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)
```

Wall time: 49.5 s

In [55]:

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

782/782 [=====] - 2s 3ms/step - loss: 1.2108 - acc: 0.6196

Out[55]:

[1.2107512950897217, 0.6195600032806396]

In []: