영화 리뷰 데이터 분류하기

학습 내용

- 10000개의 단어로 이루어진 영화 리뷰 데이터를 활용하여 긍정, 부정을 예측하는 이진분류 신경망을 구현해 본다.
- 이를 위해 데이터 전처리에 대해 배워본다.

데이터 셋

- IMDB 데이터 셋
  - 총 5만개로 이루어진 긍정 부정의 리뷰
  - 이 데이터셋은 훈련 데이터 25,000개와 테스트 데이터 25,000개로 나뉘어 있고 각각 50%는 부정, 50%는 긍정 리뷰로 구성
  - 스탠포드 대학의 앤드류 마스(Andrew Maas)가 수집한 데이터 셋

1-1 데이터 준비

In [1]:

```
from keras.datasets import imdb

(train_data, train_labels), (test_data, test_labels) = imdb.load_data(num_words=10000)
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/im
db.npz (https://storage.googleapis.com/tensorflow/tf-keras-datasets/imdb.npz)
17465344/17464789 [==============================] - 2s 0us/step


<__array_function__ internals>:5: VisibleDeprecationWarning: Creating an ndarray fro
m ragged nested sequences (which is a list-or-tuple of lists-or-tuples-or ndarrays w
ith different lengths or shapes) is deprecated. If you meant to do this, you must sp
ecify 'dtype=object' when creating the ndarray
c:\Users\toto\anaconda3\envs\tf2x\lib\site-packages\tensorflow\python\keras\datasets
\imdb.py:159: VisibleDeprecationWarning: Creating an ndarray from ragged nested sequ
ences (which is a list-or-tuple of lists-or-tuples-or ndarrays with different length
s or shapes) is deprecated. If you meant to do this, you must specify 'dtype=object'
when creating the ndarray
  x_train, y_train = np.array(xs[:idx]), np.array(labels[:idx])
c:\Users\toto\anaconda3\envs\tf2x\lib\site-packages\tensorflow\python\keras\datasets
\imdb.py:160: VisibleDeprecationWarning: Creating an ndarray from ragged nested sequ
ences (which is a list-or-tuple of lists-or-tuples-or ndarrays with different length
s or shapes) is deprecated. If you meant to do this, you must specify 'dtype=object'
when creating the ndarray
  x_test, y_test = np.array(xs[idx:]), np.array(labels[idx:])
```

- 매개변수 num_words=10000은 훈련 데이터에서 가장 자주 나타나는 단어 10,000개만 사용하겠다는 의미. 드물게 나타나는 단어는 무시
- train_data와 test_data는 리뷰
- train_labels와 test_labels (긍정 : 1, 부정 : 0)

In [2]:

```
print(train_data.shape, train_labels.shape)
print(test_data.shape, test_labels.shape)
```

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

- train_data는 여러개의 단어로 이루어진 리뷰. 리뷰 단어는 각각 매칭된 word index 값으로 이루어짐.
- train_labels는 1(긍정), 0(부정)이 됨.

In [3]:

```
# train_data의 하나(numpy)에서 10개 정도 확인해 보기
print(type(train_data[0]),  len(train_data[0]))  # 자료형과 개수
train_data[0][0:10]
```

```
<class 'list'> 218
```

Out[3]:

```
[1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65]
```

In [4]:

```
train_labels[0:5]
```

Out[4]:

```
array([1, 0, 0, 1, 0], dtype=int64)
```

10개의 리뷰에 대한 단어 인덱스 최대값 확인

- 자주 등장하는 단어 10000개로 제한

In [6]:

```
[max(sequence) for sequence in train_data][0:10] # 10개 리뷰의 각 리뷰의 단어 인덱스의 최대값
```

Out[6]:

```
[7486, 9837, 6905, 9941, 7224, 7982, 9363, 9820, 7612, 8419]
```

In [7]:

```
max([max(sequence) for sequence in train_data])
```

Out[7]:

```
9999
```

리뷰 데이터 하나를 영어 단어로 변경해 보기

- imdb.get_word_index() : 단어와 인덱스를 단어:인덱스 형태로 반환해 주는 것

In [8]:

```
# word_index는 단어와 정수 인덱스를 매핑한 딕셔너리입니다
word_index = imdb.get_word_index()

# 전체 단어:인덱스 쌍의 수
print( len(word_index) )  # 88584개
print( word_index )
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-dataset
s/imdb_word_index.json (https://storage.googleapis.com/tensorflow/tf-keras-datase
ts/imdb_word_index.json)
1646592/1641221 [==============================] - 0s 0us/step
88584
{'fawn': 34701, 'tsukino': 52006, 'nunnery': 52007, 'sonja': 16816, 'vani': 6395
1, 'woods': 1408, 'spiders': 16115, 'hanging': 2345, 'woody': 2289, 'trawling': 5
2008, "hold's": 52009, 'comically': 11307, 'localized': 40830, 'disobeying': 3056
8, "'royale": 52010, "harpo's": 40831, 'canet': 52011, 'aileen': 19313, 'acuratel
y': 52012, "diplomat's": 52013, 'rickman': 25242, 'arranged': 6746, 'rumbustiou
s': 52014, 'familiarness': 52015, "spider'": 52016, 'hahahah': 68804, "wood'": 52
017, 'transvestism': 40833, "hangin'": 34702, 'bringing': 2338, 'seamier': 40834,
'wooded': 34703, 'bravora': 52018, 'grueling': 16817, 'wooden': 1636, 'wednesda
y': 16818, "'prix": 52019, 'altagracia': 34704, 'circuitry': 52020, 'crotch': 115
85, 'busybody': 57766, "tart'n'tangy": 52021, 'burgade': 14129, 'thrace': 52023,
 "tom's": 11038, 'snuggles': 52025, 'francesco': 29114, 'complainers': 52027, 'te
mplarios': 52125, '272': 40835, '273': 52028, 'zaniacs': 52130, '275': 34706, 'co
nsenting': 27631, 'snuggled': 40836, 'inanimate': 15492, 'uality': 52030, 'bront
e': 11926, 'errors': 4010, 'dialogs': 3230, "yomada's": 52031, "madman's": 34707,
```

인덱스별 단어들

In [9]:

```
# 정수 인덱스와 단어를 매핑하도록 뒤집습니다
reverse_word_index = dict([(value, key) for (key, value) in word_index.items()])
reverse_word_index
```

```
 52009: "hold's",
 11307: 'comically',
 40830: 'localized',
 30568: 'disobeying',
 52010: "'royale",
 40831: "harpo's",
 52011: 'canet',
 19313: 'aileen',
 52012: 'acurately',
 52013: "diplomat's",
 25242: 'rickman',
 6746: 'arranged',
 52014: 'rumbustious',
 52015: 'familiarness',
 52016: "spider'",
 68804: 'hahahah',
 52017: "wood'",
 40833: 'transvestism',
 34702: "hangin'",
 2338: 'bringing',
```

인덱스 2번 단어 얻기

In [10]:

```
# reverse_word_index의 기능 확인
dir(reverse_word_index)[-11:]
```

Out[10]:

```
['clear',
 'copy',
 'fromkeys',
 'get',
 'items',
 'keys',
 'pop',
 'popitem',
 'setdefault',
 'update',
 'values']
```

In [11]:

```
# reverse_word_index.get(인덱스)  # 인덱스에 해당되는 단어가 출력
# reverse_word_index.get(인덱스, '?')  # 인덱스에 해당되는 단어가 출력되는데, 단어가 없으면 ? 출력
reverse_word_index.get(2, '?')
```

Out[11]:

```
'and'
```

첫번째 리뷰에 대해 각 단어를 리스트화 시키기

In [12]:

```
train_data[0]  # 첫번째 리뷰(숫자 인덱스)
```

```
 112,
 50,
 670,
 2,
 9,
 35,
 480,
 284,
 5,
 150,
 4,
 172,
 112,
 167,
 2,
 336,
 385,
 39,
 4,
 172,
```

In [16]:

```
# train_data[0] : 하나의 리뷰(3, 6, 2, 5, 10...) => 218단어 ...
# [reverse_word_index.get(i - 3, '?') for i in train_data[0]]
print(len( train_data[0]) )  # 첫번째 리뷰는 218개 인덱스(단어)로 이루어져 있다.
print([i for i in train_data[0]])  # train_data[0]인덱스가 for문이 돌아가면서 리스트 형태로 만들어진
```

```
218
[1, 14, 22, 16, 43, 530, 973, 1622, 1385, 65, 458, 4468, 66, 3941, 4, 173, 36, 256,
5, 25, 100, 43, 838, 112, 50, 670, 2, 9, 35, 480, 284, 5, 150, 4, 172, 112, 167, 2,
336, 385, 39, 4, 172, 4536, 1111, 17, 546, 38, 13, 447, 4, 192, 50, 16, 6, 147, 202
5, 19, 14, 22, 4, 1920, 4613, 469, 4, 22, 71, 87, 12, 16, 43, 530, 38, 76, 15, 13, 1
247, 4, 22, 17, 515, 17, 12, 16, 626, 18, 2, 5, 62, 386, 12, 8, 316, 8, 106, 5, 4, 2
223, 5244, 16, 480, 66, 3785, 33, 4, 130, 12, 16, 38, 619, 5, 25, 124, 51, 36, 135,
48, 25, 1415, 33, 6, 22, 12, 215, 28, 77, 52, 5, 14, 407, 16, 82, 2, 8, 4, 107, 117,
5952, 15, 256, 4, 2, 7, 3766, 5, 723, 36, 71, 43, 530, 476, 26, 400, 317, 46, 7, 4,
2, 1029, 13, 104, 88, 4, 381, 15, 297, 98, 32, 2071, 56, 26, 141, 6, 194, 7486, 18,
4, 226, 22, 21, 134, 476, 26, 480, 5, 144, 30, 5535, 18, 51, 36, 28, 224, 92, 25, 10
4, 4, 226, 65, 16, 38, 1334, 88, 12, 16, 283, 5, 16, 4472, 113, 103, 32, 15, 16, 534
5, 19, 178, 32]
```

In [17]:

```
# 두번째 리뷰 - 189개 인덱스(단어)로 이루어져 있고, 이에 대한 리스트를 만들어 출력해 본다.
print(len( train_data[1] ))  # 첫번째 리뷰는 218개 인덱스(단어)로 이루어져 있다.
print([i for i in train_data[1] ])  # train_data[0]인덱스가 for문이 돌아가면서 리스트 형태로 만들어
```

```
189
[1, 194, 1153, 194, 8255, 78, 228, 5, 6, 1463, 4369, 5012, 134, 26, 4, 715, 8, 118,
1634, 14, 394, 20, 13, 119, 954, 189, 102, 5, 207, 110, 3103, 21, 14, 69, 188, 8, 3
0, 23, 7, 4, 249, 126, 93, 4, 114, 9, 2300, 1523, 5, 647, 4, 116, 9, 35, 8163, 4, 22
9, 9, 340, 1322, 4, 118, 9, 4, 130, 4901, 19, 4, 1002, 5, 89, 29, 952, 46, 37, 4, 45
5, 9, 45, 43, 38, 1543, 1905, 398, 4, 1649, 26, 6853, 5, 163, 11, 3215, 2, 4, 1153,
9, 194, 775, 7, 8255, 2, 349, 2637, 148, 605, 2, 8003, 15, 123, 125, 68, 2, 6853, 1
5, 349, 165, 4362, 98, 5, 4, 228, 9, 43, 2, 1157, 15, 299, 120, 5, 120, 174, 11, 22
0, 175, 136, 50, 9, 4373, 228, 8255, 5, 2, 656, 245, 2350, 5, 4, 9837, 131, 152, 49
1, 18, 2, 32, 7464, 1212, 14, 9, 6, 371, 78, 22, 625, 64, 1382, 9, 8, 168, 145, 23,
4, 1690, 15, 16, 4, 1355, 5, 28, 6, 52, 154, 462, 33, 89, 78, 285, 16, 145, 95]
```

In [18]:

```
# [reverse_word_index.get(i - 3, '?') for i in train_data[0]]
# 의미 : 각각의 인덱스(218)에 대한 단어를 리스트 형태로 만든다.
print("첫번째 리뷰의 인덱스를 단어로 매칭시켜서 보여준 것")
print([reverse_word_index.get(i - 3, '?') for i in train_data[0]])

# ' '.join([reverse_word_index.get(i - 3, '?') for i in train_data[0]])
# 각각의 단어 리스트를 공백하나를 넣어주면서 하나의 문자열로 묶어 준것.
```

```
첫번째 리뷰의 인덱스를 단어로 매칭시켜서 보여준 것
['?', 'this', 'film', 'was', 'just', 'brilliant', 'casting', 'location', 'scenery',
'story', 'direction', "everyone's", 'really', 'suited', 'the', 'part', 'they', 'play
ed', 'and', 'you', 'could', 'just', 'imagine', 'being', 'there', 'robert', '?', 'i
s', 'an', 'amazing', 'actor', 'and', 'now', 'the', 'same', 'being', 'director', '?',
'father', 'came', 'from', 'the', 'same', 'scottish', 'island', 'as', 'myself', 'so',
'i', 'loved', 'the', 'fact', 'there', 'was', 'a', 'real', 'connection', 'with', 'thi
s', 'film', 'the', 'witty', 'remarks', 'throughout', 'the', 'film', 'were', 'great',
'it', 'was', 'just', 'brilliant', 'so', 'much', 'that', 'i', 'bought', 'the', 'fil
m', 'as', 'soon', 'as', 'it', 'was', 'released', 'for', '?', 'and', 'would', 'recomm
end', 'it', 'to', 'everyone', 'to', 'watch', 'and', 'the', 'fly', 'fishing', 'was',
'amazing', 'really', 'cried', 'at', 'the', 'end', 'it', 'was', 'so', 'sad', 'and',
'you', 'know', 'what', 'they', 'say', 'if', 'you', 'cry', 'at', 'a', 'film', 'it',
'must', 'have', 'been', 'good', 'and', 'this', 'definitely', 'was', 'also', '?', 't
o', 'the', 'two', 'little', "boy's", 'that', 'played', 'the', '?', 'of', 'norman',
'and', 'paul', 'they', 'were', 'just', 'brilliant', 'children', 'are', 'often', 'lef
t', 'out', 'of', 'the', '?', 'list', 'i', 'think', 'because', 'the', 'stars', 'tha
t', 'play', 'them', 'all', 'grown', 'up', 'are', 'such', 'a', 'big', 'profile', 'fo
r', 'the', 'whole', 'film', 'but', 'these', 'children', 'are', 'amazing', 'and', 'sh
ould', 'be', 'praised', 'for', 'what', 'they', 'have', 'done', "don't", 'you', 'thin
k', 'the', 'whole', 'story', 'was', 'so', 'lovely', 'because', 'it', 'was', 'true',
'and', 'was', "someone's", 'life', 'after', 'all', 'that', 'was', 'shared', 'with',
'us', 'all']
```

- 0, 1, 2는 '패딩', '문서 시작', '사전에 없음'을 위한 인덱스이므로 3을 빼준다.
- 첫번째 리뷰를 하나씩 인덱스를 확인 후, 매칭되는 단어를 추가. 없으면 '?'을 표시

In [19]:

```
# 리뷰를 디코딩합니다.
# 0, 1, 2는 '패딩', '문서 시작', '사전에 없음'을 위한 인덱스이므로 3을 뺍니다
decoded_review = ' '.join([reverse_word_index.get(i - 3, '?') for i in train_data[0]])
decoded_review
```

Out[19]:

```
"? this film was just brilliant casting location scenery story direction everyone's
really suited the part they played and you could just imagine being there robert ? i
s an amazing actor and now the same being director ? father came from the same scott
ish island as myself so i loved the fact there was a real connection with this film
the witty remarks throughout the film were great it was just brilliant so much that
i bought the film as soon as it was released for ? and would recommend it to everyon
e to watch and the fly fishing was amazing really cried at the end it was so sad and
you know what they say if you cry at a film it must have been good and this definite
ly was also ? to the two little boy's that played the ? of norman and paul they were
just brilliant children are often left out of the ? list i think because the stars t
hat play them all grown up are such a big profile for the whole film but these child
ren are amazing and should be praised for what they have done don't you think the wh
ole story was so lovely because it was true and was someone's life after all that wa
s shared with us all"
```

해보기

- 5번째 단어에 대해서 인덱스를 영문으로 바꿔서 표현해 보기

모델에 사용하기 위해 전처리-벡터화

리뷰 데이터는 다음과 같이 표현되었다.

- 원핫 인코딩
  - 전체 10개 단어
  - I, an, like, apple, korea, a, list, orange, pig, cat
  - 3 4 5 6 7 8 9 10 11 12
  - 0 0 0 0 0 0 0 0 0 0
- 나의 첫번째 리뷰
  - I like an apple -> (3, 5, 4, 6)
  - 0 0 0 0 0 0 0 0 0 0
  - 1 1 1 1 0 0 0 0 0 0
- 나의 두번째 리뷰
  - I like an orange -> (3, 5, 4, 10)
  - 1 1 1 0 0 0 0 1 0 0
- 나의 세번째 리뷰
  - my orange is pig -> (32000, 10, 50000, 11)
  - 0 0 0 0 0 0 0 1 1 0
- 또 다른 표현
  - 나의 첫번째 리뷰
  - I like an apple ->
  - 0 0 0 0 0 0 0 0 0 0
  - 1 1 1 1 0 0 0 0 0 0

    - (10, 0), (10, 1), (10, 2), (10,3)
- 나의 세번째 리뷰
    - my orange is pig ->
    - 0 0 0 0 0 0 0 0 0 0
    - 0 0 0 0 0 0 0 1 1 0
    - (10, 8), (10, 9)

1-2 데이터 준비

- 리스트를 원-핫 인코딩하여 0과 1의 벡터로 변환합니다.
- [3, 5]를 인덱스 3과 5의 위치는 1이고 그 외는 모두 0인 10,000차원의 벡터로 각각 변환

In [20]:

```
import numpy as np

def vectorize_sequences(sequences, dimension=10000):
    # 크기가 (len(sequences), dimension))이고 모든 원소가 0인 행렬을 만듭니다
    results = np.zeros((len(sequences), dimension))

    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.  # results[i]에서 특정 인덱스의 위치를 1로 만듭니다
    return results
```

훈련 데이터를 벡터로 변환

- 1D (25000,)에서 -> 2D (25000, 10000)으로 변경

In [21]:

```
print("변환 전 : ", train_data.shape)
X_train = vectorize_sequences(train_data)
print("변환 후 : ", X_train.shape)
X_train
```

```
변환 전 :  (25000,)
변환 후 :  (25000, 10000)
```

Out[21]:

```
array([[0., 1., 1., ..., 0., 0., 0.],
       [0., 1., 1., ..., 0., 0., 0.],
       [0., 1., 1., ..., 0., 0., 0.],
       ...,
       [0., 1., 1., ..., 0., 0., 0.],
       [0., 1., 1., ..., 0., 0., 0.],
       [0., 1., 1., ..., 0., 0., 0.]])
```

테스트 데이터를 벡터로 변환

In [22]:

```
print("변환 전 : ",test_data.shape)
X_test = vectorize_sequences(test_data)
print("변환 후 : ",X_test.shape)
X_test
```

```
변환 전 :  (25000,)
변환 후 :  (25000, 10000)
```

Out[22]:

```
array([[0., 1., 1., ..., 0., 0., 0.],
       [0., 1., 1., ..., 0., 0., 0.],
       [0., 1., 1., ..., 0., 0., 0.],
       ...,
       [0., 1., 1., ..., 0., 0., 0.],
       [0., 1., 1., ..., 0., 0., 0.],
       [0., 1., 1., ..., 0., 0., 0.]])
```

레이블을 벡터로 바꾸기

- 리스트를 배열로 만들어준다.

In [23]:

```
a = [1,2]
np.asarray(a)
```

Out[23]:

```
array([1, 2])
```

In [24]:

```
print(train_labels.shape)
print(test_labels.shape)
```

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

In [25]:

```
y_train = np.asarray(train_labels).astype('float32')
y_test = np.asarray(test_labels).astype('float32')
```

In [26]:

```
y_train
```

Out[26]:

```
array([1., 0., 0., ..., 0., 1., 0.], dtype=float32)
```

In [27]:

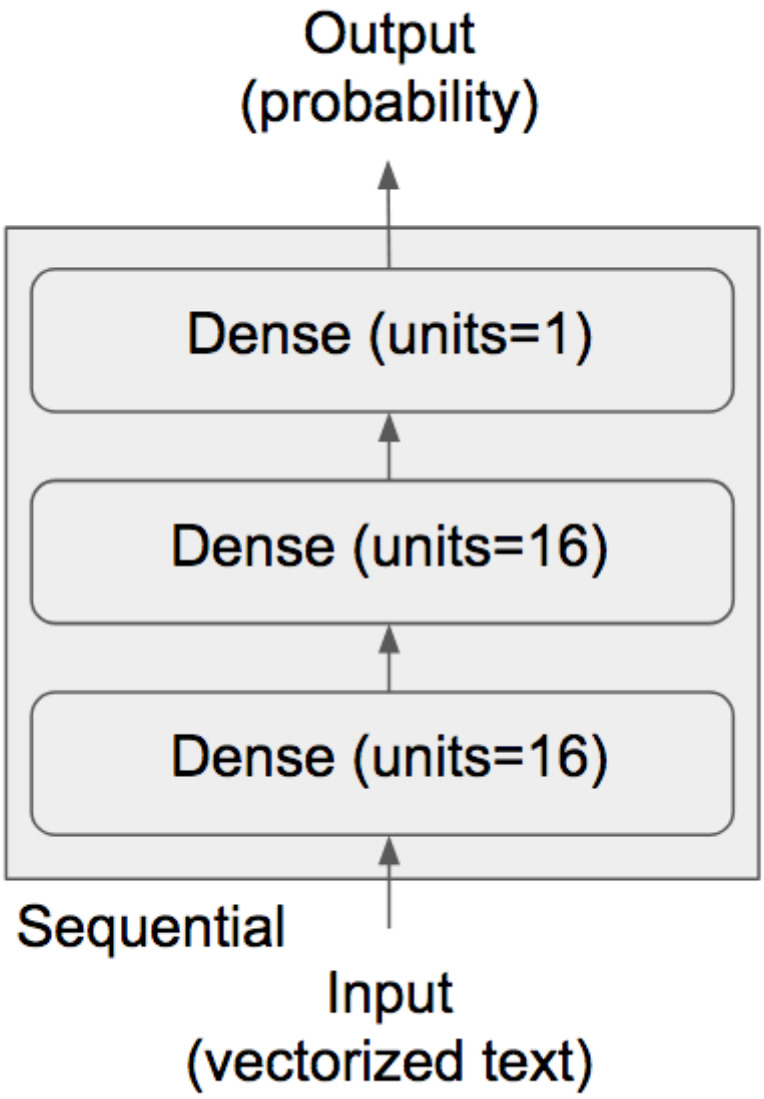
```
y_test
```

Out[27]:

```
array([0., 1., 1., ..., 0., 0., 0.], dtype=float32)
```

1-3 신경망 모델 만들기

- 얼마나 많은 은닉층 사용할 것인가?
- 각 층에 얼마나 많은 은닉 유닛을 둘 것인가?



입력 데이터가 벡터(1D)이고 레이블은 스칼라(1 또는 0)입니다.

활성화 함수(Activation)이 필요한 이유

- 선형 층을 깊게 쌓아도 여전히 하나의 선형 연산이다. 층을 여러개로 구성하는 장점이 없다.
- 이를 풍부하게 만들어 층을 깊게 만드는 장점을 살리기 위해서는 비선형성 또는 활성화 함수를 추가해야한다.
- relu는 가장 인기있는 활성화 함수중의 하나이다.

In [28]:

```
from keras import models
from keras import layers

model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

손실함수와 optimizer(최적화 함수)를 선택

In [29]:

```
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

옵티마이저의 매개변수를 설정해야 할 때,

In [30]:

```
import tensorflow as tf
import keras
print(tf.__version__)
print(keras.__version__)
```

```
2.4.1
2.4.3
```

In [31]:

```
from keras import optimizers

model.compile(optimizer=keras.optimizers.RMSprop(lr=0.01),
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

다음과 같이 가능함.

```
from keras import losses
from keras import metrics

model.compile(optimizer=optimizers.RMSprop(lr=0.001),
              loss=losses.binary_crossentropy,
              metrics=[metrics.binary_accuracy])
```

훈련 데이터 검증

- 훈련하는 동안 처음 본 데이터에 대한 모델의 정확도를 측정하기 위해 원본 훈련 데이터에서 10,000의 샘플을 떼어서 검증 세트를 만들기

In [32]:

```
X_train.shape, y_train.shape
```

Out[32]:

```
((25000, 10000), (25000,))
```

- 처음 10000개까지를 검증용, 이후의 15000개를 학습용으로 이용

In [33]:

```
X_val = X_train[:10000]
partial_X_train = X_train[10000:]

y_val = y_train[:10000]
partial_y_train = y_train[10000:]
```

In [34]:

```
history = model.fit(partial_X_train,
                    partial_y_train,
                    epochs=20,
                    batch_size=512,
                    validation_data=(X_val, y_val))
```

```
Epoch 1/20
30/30 [==============================] - 5s 114ms/step - loss: 0.7499 - accuracy: 0.
6770 - val_loss: 0.3103 - val_accuracy: 0.8787
Epoch 2/20
30/30 [==============================] - 1s 24ms/step - loss: 0.2694 - accuracy: 0.8
902 - val_loss: 0.2848 - val_accuracy: 0.8838
Epoch 3/20
30/30 [==============================] - 1s 26ms/step - loss: 0.1857 - accuracy: 0.9
254 - val_loss: 0.2951 - val_accuracy: 0.8841
Epoch 4/20
30/30 [==============================] - 1s 24ms/step - loss: 0.1401 - accuracy: 0.9
447 - val_loss: 0.2954 - val_accuracy: 0.8816
Epoch 5/20
30/30 [==============================] - 1s 26ms/step - loss: 0.0896 - accuracy: 0.9
686 - val_loss: 0.3162 - val_accuracy: 0.8808
Epoch 6/20
30/30 [==============================] - 1s 25ms/step - loss: 0.0563 - accuracy: 0.9
823 - val_loss: 0.5677 - val_accuracy: 0.8657
Epoch 7/20
30/30 [==============================] - 1s 26ms/step - loss: 0.0919 - accuracy: 0.9
673 - val_loss: 0.5825 - val_accuracy: 0.8778
Epoch 8/20
30/30 [==============================] - 1s 26ms/step - loss: 0.0805 - accuracy: 0.9
818 - val_loss: 0.5922 - val_accuracy: 0.8792
Epoch 9/20
30/30 [==============================] - 1s 25ms/step - loss: 0.0086 - accuracy: 0.9
976 - val_loss: 0.7752 - val_accuracy: 0.8784
Epoch 10/20
30/30 [==============================] - 1s 24ms/step - loss: 0.0577 - accuracy: 0.9
892 - val_loss: 0.6984 - val_accuracy: 0.8789
Epoch 11/20
30/30 [==============================] - 1s 25ms/step - loss: 0.0050 - accuracy: 0.9
984 - val_loss: 0.8724 - val_accuracy: 0.8779
Epoch 12/20
30/30 [==============================] - 1s 25ms/step - loss: 0.0031 - accuracy: 0.9
981 - val_loss: 0.9715 - val_accuracy: 0.8773
Epoch 13/20
30/30 [==============================] - 1s 24ms/step - loss: 0.0828 - accuracy: 0.9
913 - val_loss: 1.0241 - val_accuracy: 0.8752
Epoch 14/20
30/30 [==============================] - 1s 24ms/step - loss: 0.0042 - accuracy: 0.9
980 - val_loss: 1.0713 - val_accuracy: 0.8739
Epoch 15/20
30/30 [==============================] - 1s 25ms/step - loss: 0.0022 - accuracy: 0.9
990 - val_loss: 1.1854 - val_accuracy: 0.8744
Epoch 16/20
30/30 [==============================] - 1s 26ms/step - loss: 0.0015 - accuracy: 0.9
992 - val_loss: 1.2933 - val_accuracy: 0.8741
Epoch 17/20
30/30 [==============================] - 1s 24ms/step - loss: 0.0014 - accuracy: 0.9
993 - val_loss: 1.4670 - val_accuracy: 0.8749
Epoch 18/20
```

```
30/30 [==============================] - 1s 25ms/step - loss: 0.1191 - accuracy: 0.9
895 - val_loss: 1.0737 - val_accuracy: 0.8720
Epoch 19/20
30/30 [==============================] - 1s 25ms/step - loss: 0.0019 - accuracy: 0.9
992 - val_loss: 1.3230 - val_accuracy: 0.8722
Epoch 20/20
30/30 [==============================] - 1s 23ms/step - loss: 0.0013 - accuracy: 0.9
994 - val_loss: 1.4603 - val_accuracy: 0.8722
```

In [35]:

```
history_dict = history.history
history_dict.keys()
```

Out[35]:

```
dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

In [36]:

```
import matplotlib.pyplot as plt
```

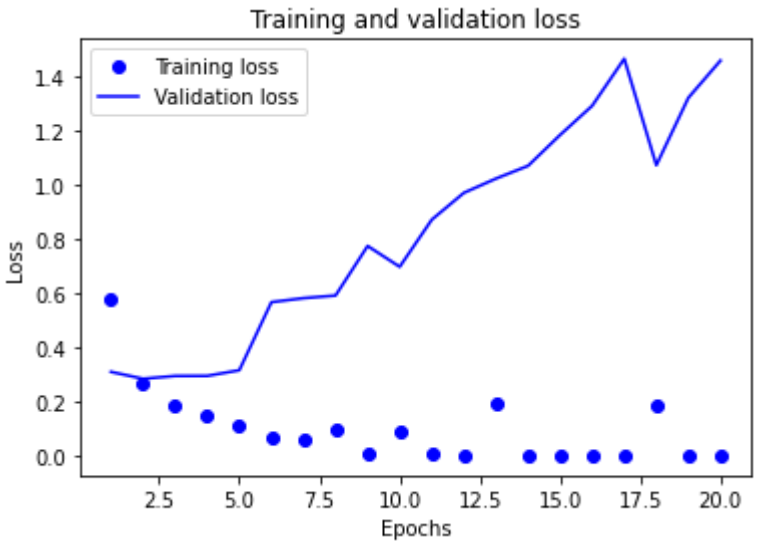
In [37]:

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

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

#  'bo' 는 파란색 점을 의미합니다
plt.plot(epochs, loss, 'bo', label='Training loss')
#  'b' 는 파란색 실선을 의미합니다
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```

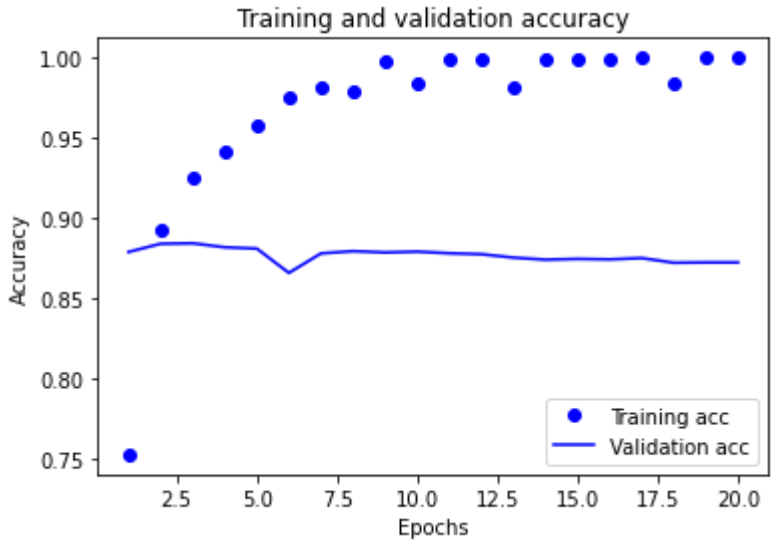


In [38]:

```
plt.clf()   # 그래프를 초기화합니다
acc = history_dict['accuracy']
val_acc = history_dict['val_accuracy']

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

plt.show()
```



- 그래프의 이해 : 훈련 손실이 에포크마다 감소하고, 훈련 정확도는 에포크마다 증가. 두번째 에포크 이후부터 훈련 데이터에 과도하게 최적화되어 훈련 데이터에 특화된 표현을 학습.

실습

- 신경망을 약간 변경한 후, 4에포크 돌리고 훈련시켜보기

처음부터 다시 새로운 신경망을 4번의 에포크 동안만 훈련하고 테스트 데이터에서 평가

- 모델을 훈련시킨 후에 이를 실전 환경에서 사용하고 싶다. predict메서드를 사용해서 어떤 리뷰가 긍정일 확률을 예측할 수 있다.

In [39]:

```
model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

model.fit(X_train, y_train, epochs=4, batch_size=512)
results = model.evaluate(X_test, y_test)
```

```
Epoch 1/4
49/49 [==============================] - 2s 16ms/step - loss: 0.5476 - accuracy: 0.7
350
Epoch 2/4
49/49 [==============================] - 1s 17ms/step - loss: 0.2664 - accuracy: 0.9
111
Epoch 3/4
49/49 [==============================] - 1s 17ms/step - loss: 0.1983 - accuracy: 0.9
291
Epoch 4/4
49/49 [==============================] - 1s 16ms/step - loss: 0.1587 - accuracy: 0.9
464
782/782 [==============================] - 2s 3ms/step - loss: 0.2996 - accuracy: 0.
8823
```

In [40]:

```
results
```

Out[40]:

```
[0.2995520234107971, 0.882319986820221]
```

1-4 예측 수행

In [41]:

```
model.predict(X_test)
```

Out[41]:

```
array([[0.15369171],
       [0.9999317 ],
       [0.72289145],
       ...,
       [0.1301493 ],
       [0.0598473 ],
       [0.50331956]], dtype=float32)
```

추가 해보기

- 1개 또는 3개의 은닉층을 사용하여 검증과 테스트 정확도 확인해 보기

- 층에 은닉 유닛을 줄이거나 늘려보기 32개, 64개 등
- binary_crossentropy대신에 mse 손실함수 사용.
- relu대신에 tanh 활성화 함수를 사용하기

추가 실습

- 01 3개의 은닉층 사용하기(노드수 동일)
- 02 2개의 은닉층 사용하기(노드수 동일)
- 03 은닉층 개수는 동일하게 하고 노드수 늘리기(32개)
- 04 은닉층 개수는 동일하게 하고 노드수 늘리기(64개)
- 05 은닉층 개수는 동일하게 하고 노드수 늘리기(128개)
- 06 tanh활성화 함수 사용하기
- 07 배치 사이즈 변경(512->128)

- 학습을 시킨이후에 그래프를 보고,
- 적절한 에폭수를 여러분이 지정하고,
- 최종결과 올리기

07 배치 사이즈 변경(512->128)

In [42]:

```
model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(optimizer=keras.optimizers.RMSprop(lr=0.001),
              loss='binary_crossentropy',
              metrics=['accuracy'])

history = model.fit(partial_X_train,
                    partial_y_train,
                    epochs=20,
                    batch_size=128,
                    validation_data=(X_val, y_val))
```

```
Epoch 1/20
118/118 [==============================] - 3s 16ms/step - loss: 0.5215 - accuracy:
0.7735 - val_loss: 0.2971 - val_accuracy: 0.8870
Epoch 2/20
118/118 [==============================] - 1s 10ms/step - loss: 0.2248 - accuracy:
0.9202 - val_loss: 0.3597 - val_accuracy: 0.8536
Epoch 3/20
118/118 [==============================] - 1s 10ms/step - loss: 0.1592 - accuracy:
0.9472 - val_loss: 0.2949 - val_accuracy: 0.8844
Epoch 4/20
118/118 [==============================] - 1s 10ms/step - loss: 0.1207 - accuracy:
0.9588 - val_loss: 0.3224 - val_accuracy: 0.8812
Epoch 5/20
118/118 [==============================] - 1s 10ms/step - loss: 0.0929 - accuracy:
0.9686 - val_loss: 0.3733 - val_accuracy: 0.8730
Epoch 6/20
118/118 [==============================] - 1s 10ms/step - loss: 0.0726 - accuracy:
0.9752 - val_loss: 0.4033 - val_accuracy: 0.8714
Epoch 7/20
118/118 [==============================] - 1s 10ms/step - loss: 0.0563 - accuracy:
0.9816 - val_loss: 0.4424 - val_accuracy: 0.8738
Epoch 8/20
118/118 [==============================] - 1s 10ms/step - loss: 0.0413 - accuracy:
0.9885 - val_loss: 0.4936 - val_accuracy: 0.8702
Epoch 9/20
118/118 [==============================] - 1s 10ms/step - loss: 0.0285 - accuracy:
0.9921 - val_loss: 0.5622 - val_accuracy: 0.8691
Epoch 10/20
118/118 [==============================] - 1s 10ms/step - loss: 0.0223 - accuracy:
0.9945 - val_loss: 0.6271 - val_accuracy: 0.8672
Epoch 11/20
118/118 [==============================] - 1s 10ms/step - loss: 0.0143 - accuracy:
0.9968 - val_loss: 0.7049 - val_accuracy: 0.8599
Epoch 12/20
118/118 [==============================] - 1s 10ms/step - loss: 0.0092 - accuracy:
0.9982 - val_loss: 0.7902 - val_accuracy: 0.8626
Epoch 13/20
118/118 [==============================] - 1s 10ms/step - loss: 0.0069 - accuracy:
0.9988 - val_loss: 0.9008 - val_accuracy: 0.8609
Epoch 14/20
118/118 [==============================] - 1s 10ms/step - loss: 0.0035 - accuracy:
0.9995 - val_loss: 0.9988 - val_accuracy: 0.8593
Epoch 15/20
```

```
118/118 [==============================] - 1s 11ms/step - loss: 0.0019 - accuracy:
0.9998 - val_loss: 1.0777 - val_accuracy: 0.8599
Epoch 16/20
118/118 [==============================] - 1s 11ms/step - loss: 8.5888e-04 - accurac
y: 1.0000 - val_loss: 1.2156 - val_accuracy: 0.8568
Epoch 17/20
118/118 [==============================] - 1s 11ms/step - loss: 9.8284e-04 - accurac
y: 0.9998 - val_loss: 1.2755 - val_accuracy: 0.8556
Epoch 18/20
118/118 [==============================] - 1s 11ms/step - loss: 4.0539e-04 - accurac
y: 1.0000 - val_loss: 1.4033 - val_accuracy: 0.8549
Epoch 19/20
118/118 [==============================] - 1s 11ms/step - loss: 1.6920e-04 - accurac
y: 1.0000 - val_loss: 1.5176 - val_accuracy: 0.8546
Epoch 20/20
118/118 [==============================] - 1s 11ms/step - loss: 7.2228e-05 - accurac
y: 1.0000 - val_loss: 1.6156 - val_accuracy: 0.8565
```

In [43]:

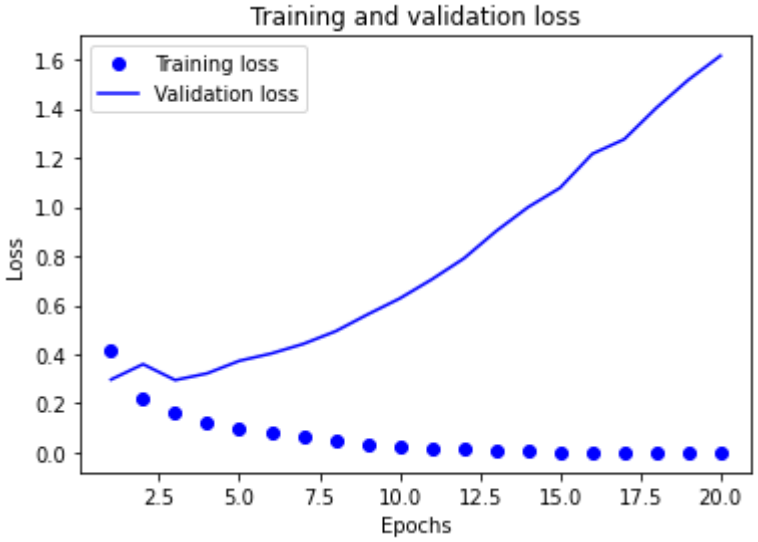
```
import matplotlib.pyplot as plt

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

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

#  'bo' 는 파란색 점을 의미합니다
plt.plot(epochs, loss, 'bo', label='Training loss')
#  'b' 는 파란색 실선을 의미합니다
plt.plot(epochs, val_loss, 'b', label='Validation loss')
plt.title('Training and validation loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()

plt.show()
```

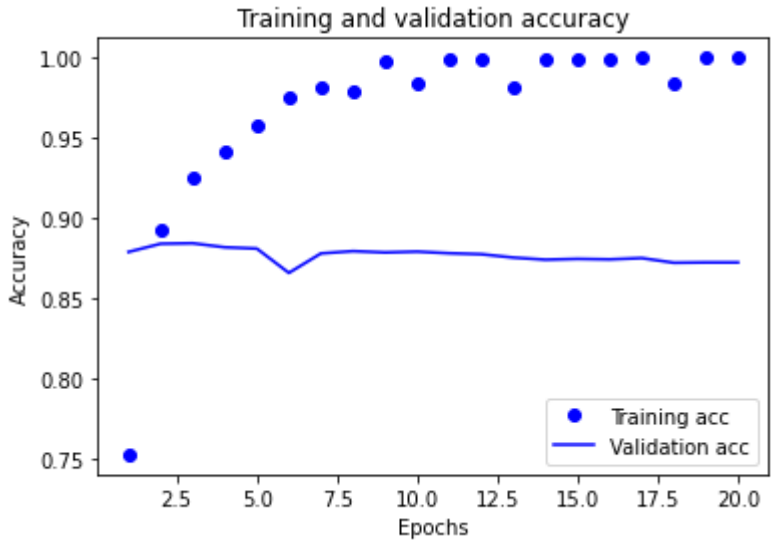


In [44]:

```
plt.clf()   # 그래프를 초기화합니다
acc = history_dict['accuracy']
val_acc = history_dict['val_accuracy']

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

plt.show()
```



In [45]:

```
model = models.Sequential()
model.add(layers.Dense(16, activation='relu', input_shape=(10000,)))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(optimizer=keras.optimizers.RMSprop(lr=0.001),
              loss='binary_crossentropy',
              metrics=['accuracy'])

model.fit(X_train, y_train, epochs=4, batch_size=128)
results = model.evaluate(X_test, y_test)
print("에폭수 : ", epochs, "배치사이즈 :", 128)
print("loss , accuracy", results)
```

```
Epoch 1/4
196/196 [==============================] - 3s 7ms/step - loss: 0.4391 - accuracy: 0.
8143
Epoch 2/4
196/196 [==============================] - 1s 7ms/step - loss: 0.2032 - accuracy: 0.
9238
Epoch 3/4
196/196 [==============================] - 1s 7ms/step - loss: 0.1680 - accuracy: 0.
9403
Epoch 4/4
196/196 [==============================] - 1s 7ms/step - loss: 0.1338 - accuracy: 0.
9527
782/782 [==============================] - 2s 3ms/step - loss: 0.3673 - accuracy: 0.
8672
에폭수 :  range(1, 21) 배치사이즈 : 128
loss , accuracy [0.367267370223999, 0.8672000169754028]
```

확인(loss, accuracy)

- 07 배치 사이즈 512 -> 128
  - 2.65, 0.498
- 07 배치 사이즈 512 -> 64
  - 2.087, 0.4965
- 06 tanh활성화 함수 사용하기
  - relu -> tanh 1.53, 0.49
- 학습률 : 0.01 -> 0.001, epochs(4), batch_size = 128
  - 2.545, 0.496
- 학습률 : 0.01 -> 0.001, epochs(8), batch_size = 128
  - 4.056, 0.497
- 에폭변경 : 4epochs, batch_size = 512, 학습률 : 0.01, 노드 수: 16->128
  - loss, accuracy : 2.85, 0.496
- 에폭변경 : 4epochs -> 6epochs, batch_size = 512, 학습률 : 0.01
  - loss, accuracy : 0.2961, 0.881
- 에폭변경 : 4epochs -> 6epochs, batch_size = 128, 학습률 : 0.01
  - loss, accuracy : 10.03, 0.4965
- 에폭변경 : 4epochs -> 8epochs, batch_size = 512, 학습률 : 0.01, 노드 수: 16->64
  - loss, accuracy : 3.714, 0.496

정리해보기

- 원본 데이터를 신경망에 주입하기 위해서 꽤 많은 전처리가 필요.
- 이진 분류의 문제에서 네트워크는 하나의 유닛과 sigmoid 활성화 함수를 가진 Dense층으로 끝나야 함.
  - 이 신경망의 출력은 확률을 나타내는 0과 1 사이의 스칼라 값이다.
  - 이진 분류 문제에서 이런 스칼라 시그모이드 출력에 대해 사용할 손실함수는 binary_crossentropy
- 훈련 데이터에 대해 성능이 향상됨에 따라 신경망은 과대적합이 시작된다. 따라서 훈련 세트 이외의 데이터에서 성능 모니터링을 해야 함.

In [ ]: