딥러닝 이진 분류

감성분석3 – 딥러닝 기반

최 석 재

lingua@naver.com

딥러닝 이진분류 *Binary Classification*

구글 드라이브와 연결

```
# from google.colab import auth
# auth.authenticate_user()
```

- from google.colab import drive
- drive.mount('/content/gdrive')

형태소분석기 설치

- !apt-get update
- !apt-get install g++ openjdk-8-jdk

- !pip install JPype1
- !pip install rhinoMorph

경로 변경

파일이 있는 곳으로 경로를 변경한다

- !cd /content/gdrive/My₩ Drive/pytest/

형태소 분석된 데이터 로딩

```
def read_data(filename, encoding='cp949'):                    # 읽기 함수 정의
    with open(filename, 'r', encoding=encoding) as f:
        data = [line.split('₩t') for line in f.read().splitlines()]
        data = data[1:]                    # 첫 행은 헤더(id document label)일 수 있으므로 제외
    return data

def write_data(data, filename, encoding='cp949'):             # 쓰기 함수 정의
    with open(filename, 'w', encoding=encoding) as f:
        f.write(data)

data = read_data('ratings_morphed.txt' , encoding='cp949' )

print(type(data))
print(len(data))
print(len(data[0]))
print(data[0])
```

```
<class 'list'>
197559
3
['8132799', '디자인 배우 학생 외국 디자이너 일구 전통 통하 발전 문화 산업 부럽 사실 우리나라
```

데이터 줄이기

- import random
- import math
- import numpy as np
- random.shuffle(data) # data를 랜덤하게 섞음

- part_num = math.floor(len(data) * 1/3) # data의 1/3을 정수로 얻음
- data = data[:part_num] # 앞에서부터 1/3 크기의 데이터만 선택
- print(len(data)) # 65853

무료 Colab에서 실행하기에는 데이터가 너무 많아 1/3로 줄인다

데이터 분리

훈련데이터와 테스트데이터 분리

- data_text = [line[1] for line in data] # 데이터 본문
- data_senti = [line[2] for line in data] # 데이터 긍부정 부분

- from sklearn.model_selection import train_test_split
- train_data_text, test_data_text, train_data_senti, test_data_senti = train_test_split(data_text, data_senti, stratify=data_senti)

분리된 데이터 확인

Counter 클래스를 이용해 train과 test 데이터의 비율을 확인한다

- from collections import Counter
- train_data_senti_freq = Counter(train_data_senti)
- print('train_data_senti_freq:', train_data_senti_freq)

- test_data_senti_freq = Counter(test_data_senti)
- print('test_data_senti_freq:', test_data_senti_freq)

```
train_data_senti_freq: Counter({'0': 24849, '1': 24540})
test_data_senti_freq: Counter({'0': 8284, '1': 8180})
```

데이터의 길이 통계

- import numpy as np
- text_len = [len(line.split(' ')) for line in train_data_text] # 단어 분리 후 개수 저장
- print("최소길이: ", np.min(text_len)) # 1
- print("최대길이: ", np.max(text_len)) # 70
- print("평균길이: ", np.round(np.mean(text_len), 1)) # 8.8
- print("중위수길이: ", np.median(text_len)) # 7.7
- print("구간별 최대 길이: ", np.percentile(text_len, [0, 25, 50, 75, 90, 100]))

```
[ 1.  4.  7. 11. 19. 70.]
```

※ 사용할 단어 개수는 90%를 담을 수 있는 20개로 한다

- print("최소길이 문장: ", train_data_text[np.argmin(text_len)])
- print("최대길이 문장: ", train_data_text[np.argmax(text_len)])

```
최소길이 문장:  정말
최대길이 문장:  ㅋㅋ ㅋㅋ ㅋㅋ ㅋㅋ ㅋㅋ ㅋㅋ ㅋㅋ ㅋㅋ ㅋㅋ ㅋㅋ ㅋㅋ ㅋㅋ ㅋㅋ ㅋㅋ ㅋㅋ ㅋㅋ ㅋㅋ ㅋㅋ ㅋㅋ ㅋㅋ ㅋㅋ ㅋㅋ ㅋㅋ ㅋㅋ
```

Data Tokenizing

단어에 숫자 기호를 배당하는 Tokenizing

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

- max_words = 10000 # 데이터셋에서 가장 빈도 높은 10,000 개의 단어만 사용
- maxlen = 20 # 20개 이후의 단어는 버려 각 문장의 길이를 고정

- tokenizer = Tokenizer(num_words=max_words) # 상위빈도 10,000 개 단어를 추려내는 Tokenizer 객체 생성
- tokenizer.fit_on_texts(train_data_text) # 전체 단어를 대상으로 인덱스를 구축한다
- word_index = tokenizer.word_index # 단어 인덱스를 가져온다

Tokenizer 결과 확인

Tokenizing 결과 확인

- print('전체에서 %s개의 고유한 토큰을 찾았습니다.' % len(word_index))
- print('word_index type: ', type(word_index))
- print('word_index: ', word_index)

```
전체에서 21965개의 고유한 토큰을 찾았습니다.
word_index type:  <class 'dict'>
word_index:  {'영화': 1, '하': 2, '보': 3, '없': 4, 'ㅋㅋ': 5, '재미있': 6, '좋': 7, '너무': 8, '되': 9, '있': 10,
```

Data Sequencing

텍스트를 숫자로 변환
상위 빈도 10,000(max_words)개의 단어만 추출하여 word_index의 숫자 리스트로 변환

- data = tokenizer.texts_to_sequences(train_data_text) # 데이터에 Tokenizer 적용

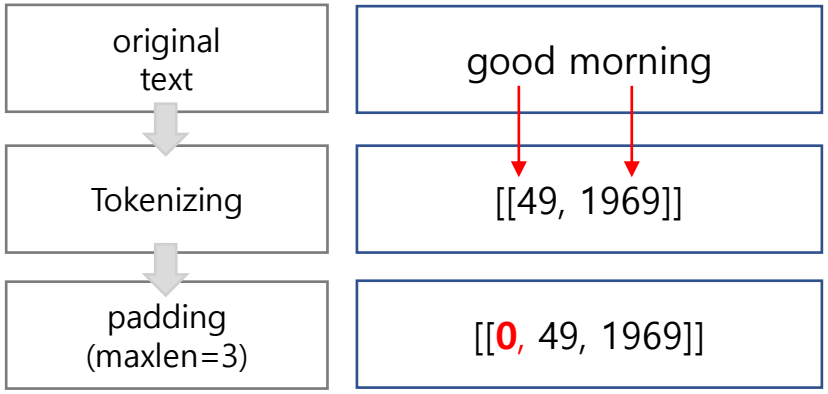
- print('data 0:', data[0])
- print('texts 0:', train_data_text[0])

```
data 0: [278, 83, 56, 663, 83, 666, 59, 2, 1118, 12, 72]
texts 0: 엔딩 마음 들 한편 마음 걸리 완전 하 해피엔딩 같 느낌
```

※ 토큰으로 선정된 각 단어에 대하여 index가 배당된다
랜덤 추출되었기 때문에 결과는 매번 다르다

Data Padding

Padding은 데이터의 길이를 고정시켜 준다
지정된 길이에 모자라는 것은 0으로 채우고, 넘치는 것은 잘라낸다
기본값으로 단어의 선택은 뒤에서부터 한다



- data = pad_sequences(data, maxlen=maxlen)

- print('data:', data)
- print('data 0:', data[0])
- print('data 0의 길이:', len(data[0]))

```
data: [[   0    0    0 ... 1118   12   72]
 [   0    0    0 ...    0    0  296]
 [   0    0    0 ...   54   66    7]
 ...
 [   0    0    0 ...   76  810 2961]
 [   0    0    0 ...   34   18   16]
 [   0    0    0 ...   26  686   97]]
data 0: [   0    0    0    0    0    0    0    0    0  278   83   56  663   83
  666   59    2 1118   12   72]
data 0의 길이: 20
```

Data Type 확인

- print(type(train_data_text))
- print(type(data))
- print(data.shape)

```
<class 'list'>
<class 'numpy.ndarray'>
(49389, 20)
```

One-Hot Encoding 연습

one-hot encoding은 모든 숫자를 0과 1로만 만든다

숫자로 치환된 라벨이 다중분류에서 올바른 곳에 표시될 수 있게 한다

- sample = [[5, 6, 7], [8, 9, 10]]

- arr = np.zeros((len(sample), 10+1)) # "10"은 11번째에 들어가게 되므로 11개의 공간을 가진 np.array를 만든다 (패딩 0 고려)

- for i, seq in enumerate(sample): # 리스트가 2개이므로 i는 총 2회(0, 1) 반복되며,

  arr[i, seq] = 1. # 각 i에서 리스트의 number가 가리키는 곳의 배열 위치에 1을 기록한다

  [0, [5, 6, 7]]
  [1, [8, 9, 10]]
  [행, 열]

- arr

```
In[108]: arr                                    해당 열 위치에 1을 입력한다
Out[108]:
array([[0., 0., 0., 0., 0., 1., 1., 1., 0., 0., 0.],   <- i == 0
       [0., 0., 0., 0., 0., 0., 0., 0., 1., 1., 1.]])  <- i == 1
        0   1   2   3   4   5   6   7   8   9   10
```

One-Hot Encoding

```
def to_one_hot(sequences, dimension):
    results = np.zeros((len(sequences), dimension))
    for i, sequence in enumerate(sequences):
        results[i, sequence] = 1.
    return results
```

```
data = to_one_hot(data, dimension=max_words)
labels = np.asarray(train_data_senti).astype('float32')
```

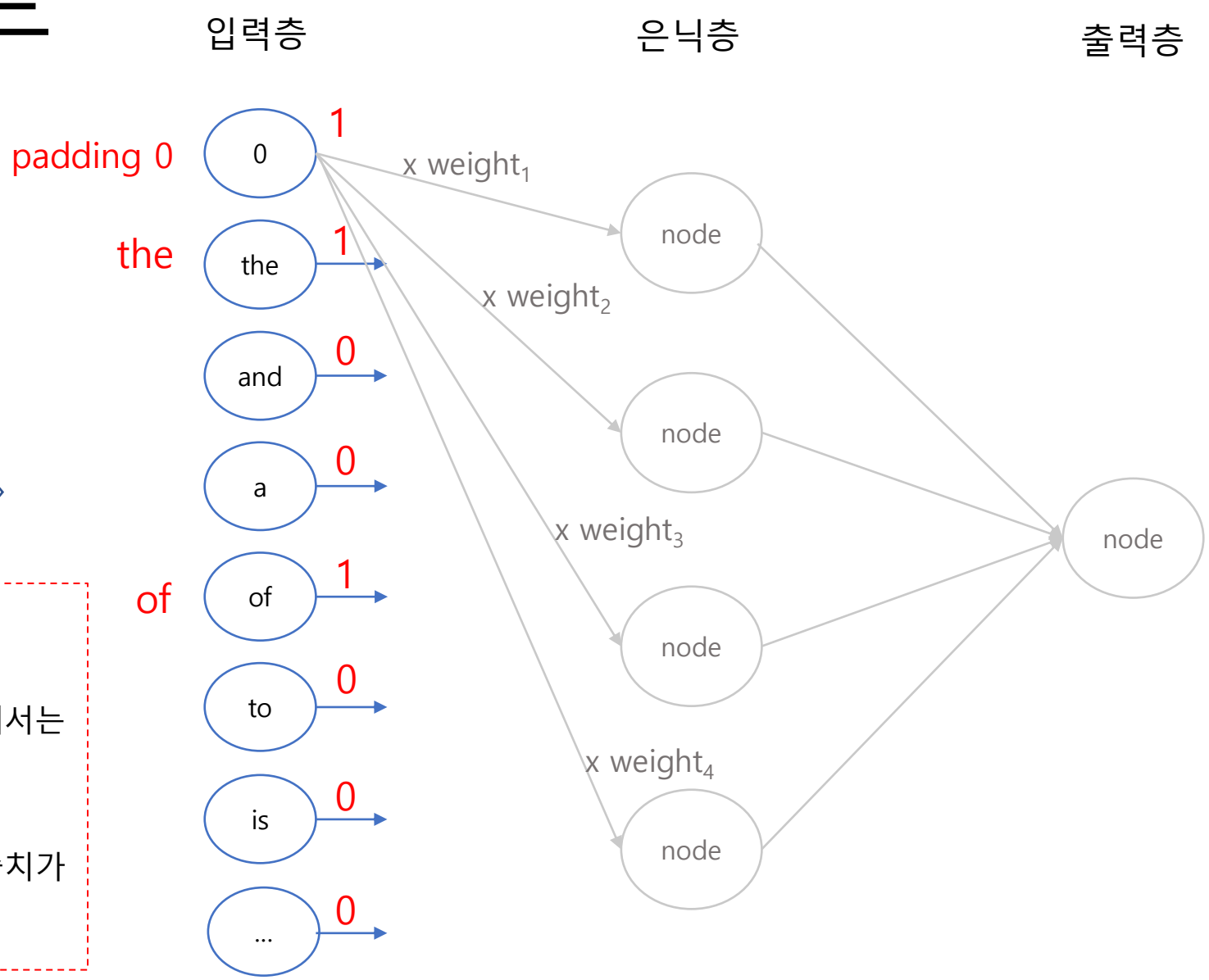
One-Hot Encoding 결과 확인

- print('data:', data)
- print(len(data[0]))
- print('data [0][0:100]:', data[0][0:100])

※ dimension=10000이므로 각 행은 10,000개를 가지고 있다

```
data: [[1. 0. 1. ... 0. 0. 0.]
 [1. 0. 0. ... 0. 0. 0.]
 [1. 0. 1. ... 0. 0. 0.]
 ...
 [1. 0. 0. ... 0. 0. 0.]
 [1. 0. 1. ... 0. 0. 0.]
 [1. 0. 0. ... 0. 0. 0.]]
10000
data [0][0:100]: [1. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 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. 1. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
 0. 0. 0. 0.]
```

입력부 노드



"the gate of the house"

※ 이상의 결과로 신경망의 입력부 노드는
위와 같이 원-핫-인코딩 결과로 시작된다

입력층의 노드는 총 10,000개이며 위 예에서는
5개 종류의 어휘를 가지고 있으므로
이들 중 5개 노드만 1이 된다

이들 각각은 은닉층의 노드에 각각의 가중치가
곱해져 전달된다
나머지는 0이므로 값이 전달되지 않는다

```
In[118]: word_index
Out[118]:
{'the': 1,
 'and': 2,
 'a': 3,
 'of': 4,
 'to': 5,
 'is': 6,
 'br': 7,
 'in': 8,
 'it': 9,
 'i': 10,
 'this': 11,
 'that': 12,
 'was': 13,
 'as': 14,
 'for': 15,
 'with': 16,
 'movie': 17,
```

Data 확인

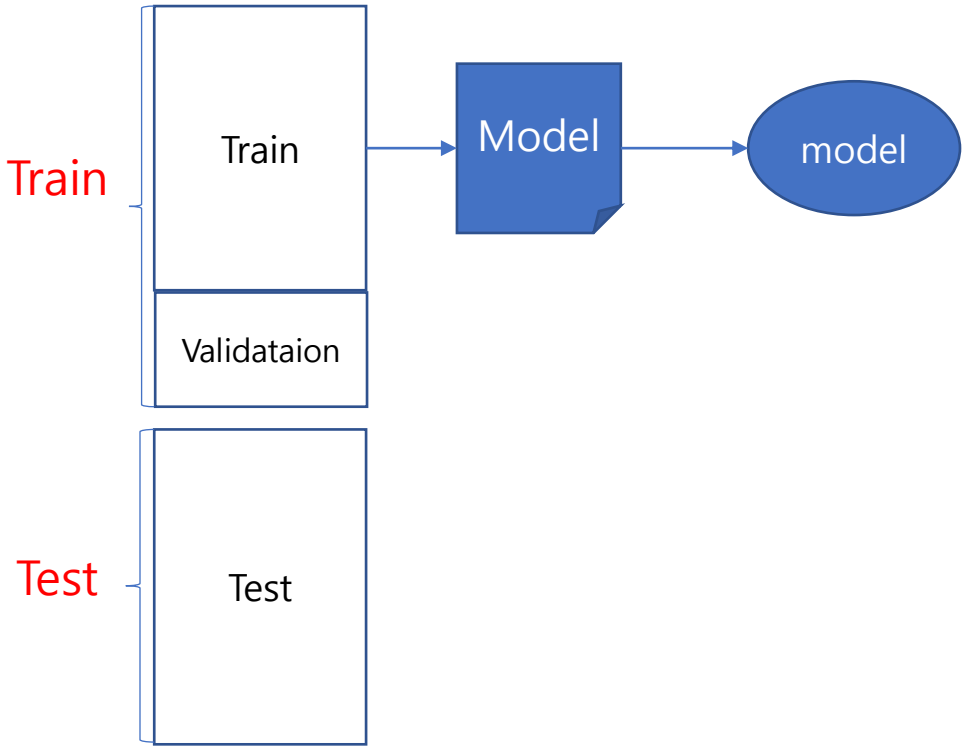
- print(type(train_data_text))
- print(type(data))
- print(data.shape)

- print('데이터 텐서의 차원:', data.ndim)
- print('레이블 텐서의 차원:', labels.ndim)

- print('데이터 텐서의 크기:', data.shape)
- print('레이블 텐서의 크기:', labels.shape)

```
<class 'list'>
<class 'numpy.ndarray'>
(49389, 10000)
데이터 텐서의 차원: 2
레이블 텐서의 차원: 1
데이터 텐서의 크기: (49389, 10000)
레이블 텐서의 크기: (49389,)
```

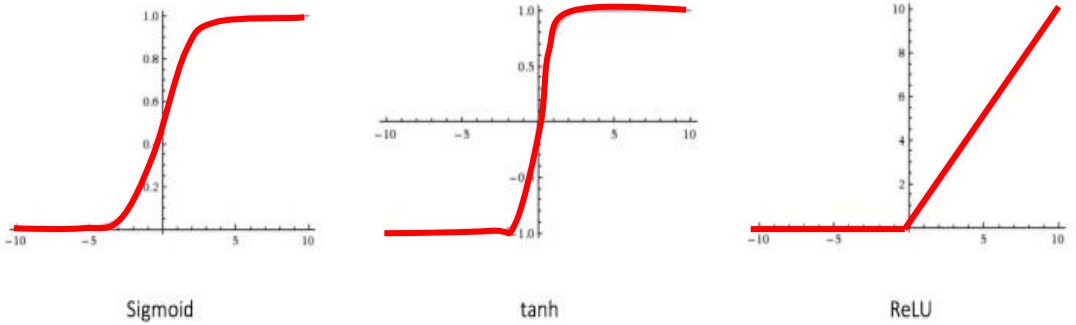
훈련데이터에서 검증데이터 분리



- validation_ratio = 0.3 # 30%는 검증데이터로 사용한다. 나머지는 훈련데이터
- validation_len = math.floor(len(train_data_text) * validation_ratio)

- x_train = data[validation_len:] # 훈련데이터의 70%는 훈련데이터
- y_train = labels[validation_len:] # 훈련데이터의 70%는 훈련데이터 Label
- x_val = data[:validation_len] # 훈련데이터의 30%는 검증데이터
- y_val = labels[:validation_len] # 훈련데이터의 30%는 검증데이터 Label

모델 설계



- from keras.models import Sequential
- from keras.layers import Dense

- model = Sequential() # 모델 새로 정의

은닉노드의 수 / 활성화 함수 음을 0으로 만드는 relu / 입력층에는 10,000 개 단어가 온다

- model.add(Dense(64, activation='relu', input_shape=(max_words,))) # 첫 번째 은닉층
- model.add(Dense(32, activation='relu')) # 두 번째 은닉층
- model.add(Dense(1, activation='sigmoid'))

```
# 출력층
# 이진분류 문제이고 신경망의 출력이 확률로 나와야 하므로,
# 0~1로 출력하는 sigmoid를 택하고, 노드는 1개로 하였다
# 다중분류에서는 softmax를 사용한다
```

※ Dense 함수에서 각 층의 옵션을 구체적으로 결정할 수 있다
※ activation은 다음 층으로 값을 넘기는 방법

모델 요약 출력

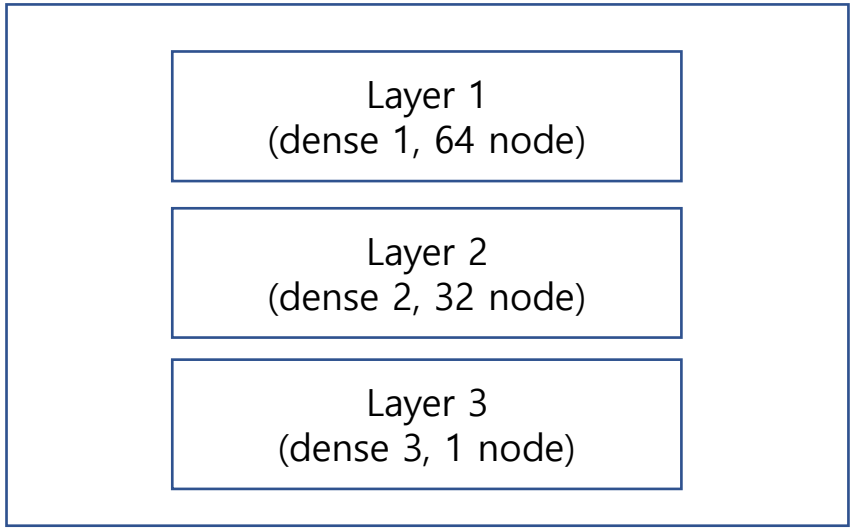
- model.summary()

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 64)                640064

 dense_1 (Dense)             (None, 32)                2080

 dense_2 (Dense)             (None, 1)                 33

=================================================================
Total params: 642,177
Trainable params: 642,177
Non-trainable params: 0
_________________________________________________________________
```



Compile & Train Model

모델 컴파일
가중치 업데이트 방법은 RMSprop을 사용하였다. 이동평균의 방법을 도입하여 조절해간다
신경망의 출력이 확률이므로 crossentropy를 사용한다
crossentropy는 원본의 확률 분포와 예측의 확률 분포를 측정하여 조절해 간다
또한 이진 분류이므로 binary_crossentropy를 사용한다

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

- loss는 오차값을 계산하는 방법
- optimizer는 오차를 줄여나가는 방법

모델 훈련
32개씩 배치를 만들어 5번의 epoch로 훈련한다. 보통 32개에서 시작하여 512개까지 중에서 찾는다
훈련 데이터로 훈련하고, 검증 데이터로 검증한다
반환값의 history는 훈련하는 동안 발생한 모든 정보를 담고 있는 딕셔너리이다

- history = model.fit(x_train, y_train, epochs=5, batch_size=32, validation_data=(x_val, y_val))
- history_dict = history.history

```
Epoch 1/5
1081/1081 [==============================] - 15s 12ms/step - loss: 0.4357 - acc: 0.7999 - val_loss: 0.4115 - val_acc: 0.8126
Epoch 2/5
1081/1081 [==============================] - 12s 11ms/step - loss: 0.3691 - acc: 0.8390 - val_loss: 0.4050 - val_acc: 0.8208
Epoch 3/5
1081/1081 [==============================] - 12s 11ms/step - loss: 0.3520 - acc: 0.8488 - val_loss: 0.4041 - val_acc: 0.8201
Epoch 4/5
1081/1081 [==============================] - 12s 11ms/step - loss: 0.3389 - acc: 0.8573 - val_loss: 0.4001 - val_acc: 0.8231
Epoch 5/5
1081/1081 [==============================] - 12s 11ms/step - loss: 0.3261 - acc: 0.8652 - val_loss: 0.4063 - val_acc: 0.8224
```

Save Model

경로 변경
- %cd /content/gdrive/My Drive/pytest/

multidimensional numpy arrays를 저장할 수 있는 h5 file(HDF) 포맷으로 저장
- model.save('text_binary_model.h5')

훈련데이터에서 사용된 상위빈도 10,000개의 단어로 된 Tokenizer 저장
새로 입력되는 문장에서도 같은 단어가 추출되게 한다
- import pickle
- with open('text_binary_tokenizer.pickle', 'wb') as handle:
- pickle.dump(tokenizer, handle, protocol=pickle.HIGHEST_PROTOCOL)

모델 성능 확인

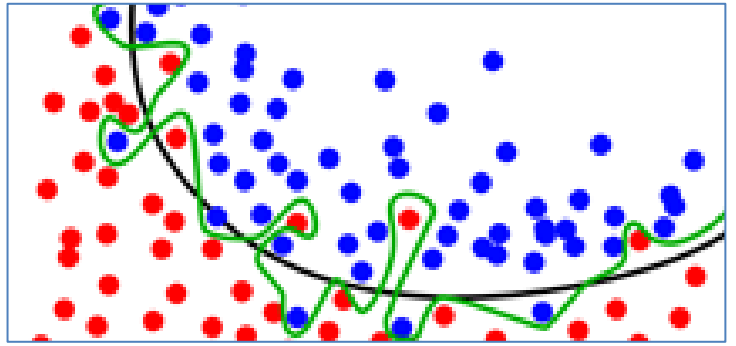
Accuracy & Loss 확인
history 딕셔너리 안에 있는 정확도와 손실값을 가져와 본다

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

- print('Train accuracy of each epoch:', np.round(acc, 3))
- print('Validation accuracy of each epoch:', np.round(val_acc, 3))

```
Train accuracy of each epoch: [0.8   0.839 0.849 0.857 0.865]
Validation accuracy of each epoch: [0.813 0.821 0.82  0.823 0.822]
```

Plotting Accuracy

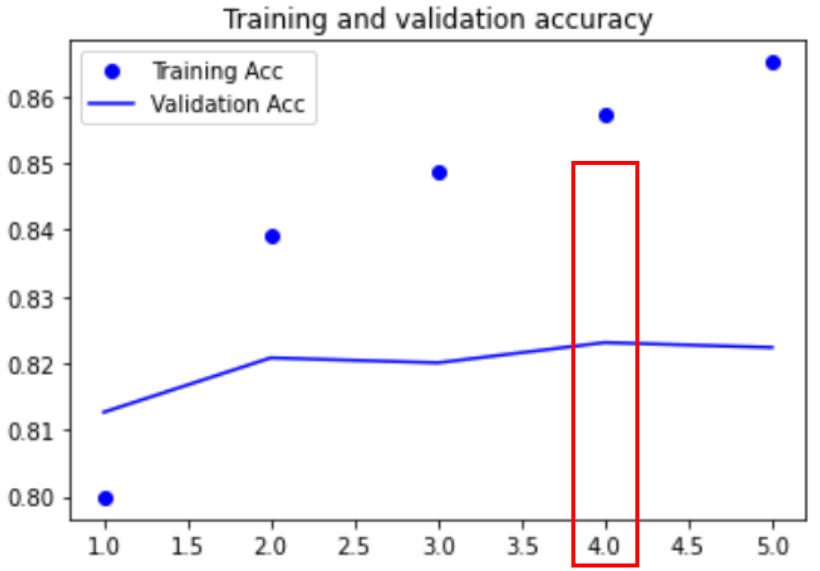


- import matplotlib.pyplot as plt

훈련데이터의 정확도에 비해 검증데이터의 정확도는 낮게 나타난다
epoch가 높아지면 모델은 훈련데이터에 매우 민감해져 오히려 새로운 데이터를 잘 못 맞춘다

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

- 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.show()



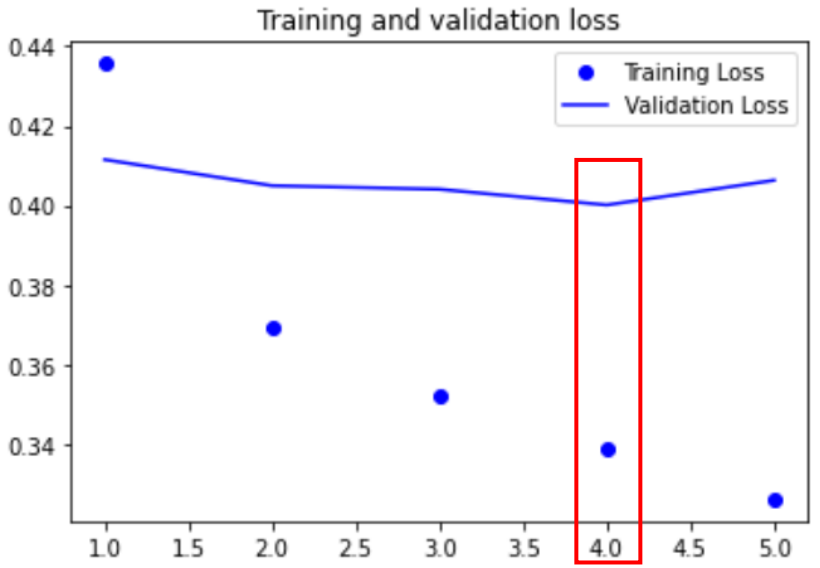
Plotting Loss

- plt.figure() # 새로운 그림을 그린다

훈련데이터의 손실값은 낮아지나, 검증데이터의 손실값은 높아진다
손실값은 오류값을 말한다. 예측과 정답의 차이를 거리 계산으로 구한 값이다

- 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()



Load Model

- import os
- from keras.models import load_model

- filepath = '/content/gdrive/My Drive/pytest/'
- os.chdir(filepath)
- print("Current Directory:", os.getcwd())

- loaded_model = load_model('text_binary_model.h5')
- print("model loaded:", loaded_model)

- with open('text_binary_tokenizer.pickle', 'rb') as handle:
- loaded_tokenizer = pickle.load(handle)

테스트 데이터 준비

- print(test_data_text)
- print(test_data_senti)

```
['좋 영화', '밑 의분 말 좀 빌리 제목 아내 유혹 루비 반지', '진짜 맛깔 날 불량 식품 같 작품',
['1', '0', '1', '0', '1', '0', '1', '1', '0', '1', '0', '0', '1', '0', '1', '0', '1', '1',
```

테스트 데이터 Sequencing

문자열을 word_index의 숫자 리스트로 변환

- data = loaded_tokenizer.texts_to_sequences(test_data_text)

padding으로 문자열의 길이를 고정시킨다

- data = pad_sequences(data, maxlen=maxlen)

test 데이터를 원-핫 인코딩한다

- x_test = to_one_hot(data, dimension=max_words)

test_data_senti를 list에서 넘파이 배열로 변환

- y_test = np.asarray(test_data_senti).astype('float32')

테스트 데이터 평가

모델에 분류할 데이터와 그 정답을 같이 넣어준다

- test_eval = loaded_model.evaluate(x_test, y_test)

모델이 분류한 결과와 입력된 정답을 비교한 결과

- print('prediction model loss & acc:', test_eval)

```
515/515 [==============================] - 2s 3ms/step - loss: 0.3918 - acc: 0.8308
prediction model loss & acc: [0.39180219173431396, 0.830782294273376 5]
```

1개 데이터 예측

형태소분석을 포함하여 이제까지의 과정을 모두 진행해주어야 한다

- text = ["재미있게 잘 봤습니다"] # 데이터를 list 타입으로 만든다

- import rhinoMorph
- rn = rhinoMorph.startRhino()

리스트 컴프리헨션으로 실질형태소만을 리스트로 가져온다

- text=[rhinoMorph.onlyMorph_list(rn, sentence, pos=['NNG', 'NNP', 'NP', 'VV', 'VA', 'XR', 'IC', 'MM', 'MAG', 'MAJ'], eomi=False) for sentence in text]
- print('형태소 분석 결과:', text)

- data = loaded_tokenizer.texts_to_sequences(text)
- data = pad_sequences(data, maxlen=maxlen)
- x_test = to_one_hot(data, dimension=max_words)

- prediction = loaded_model.predict(x_test)
- print("Result:", prediction)

[[0.9262649]]. 1(긍정)일 확률이 93%

```
filepath:  /usr/local/lib/python3.7/dist-packages
classpath:  /usr/local/lib/python3.7/dist-packages/rhinoMorph/lib/rhino.jar
RHINO started!
형태소 분석 결과: [['재미있', '잘', '보']]
Result: [[0.9262649]]
```