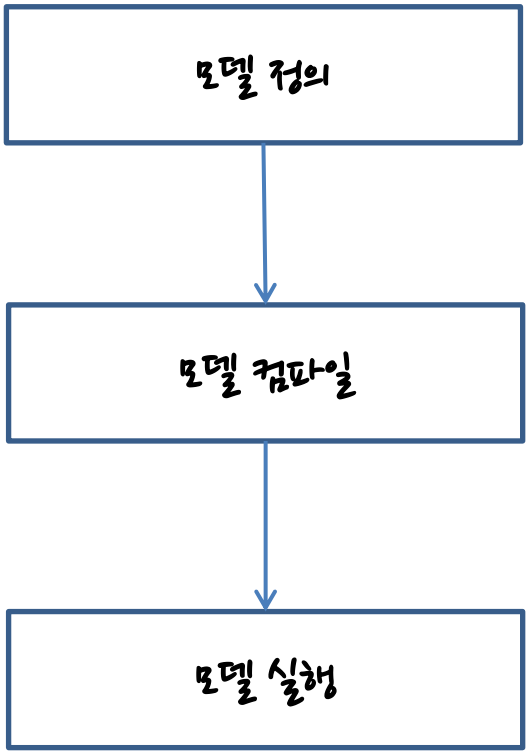
기본기 다지기

2021

■ 과정



- model이라는 함수를 선언하며 시작
- 입력층, 은닉층, 출력층 설계
- 각 층마다 활성화 함수를 설정

- 앞서 지정한 모델이 효과적으로 구현될 수 있게 여러 가지 환경을 설정해 주는 과정
- 오차 함수, 최적화 함수 설정

- 반복 횟수(Epoch) 설정
- 한번에 처리할 샘플 개수(Batch size) 설정

■ 폐암 환자의 생존율 예측 사례

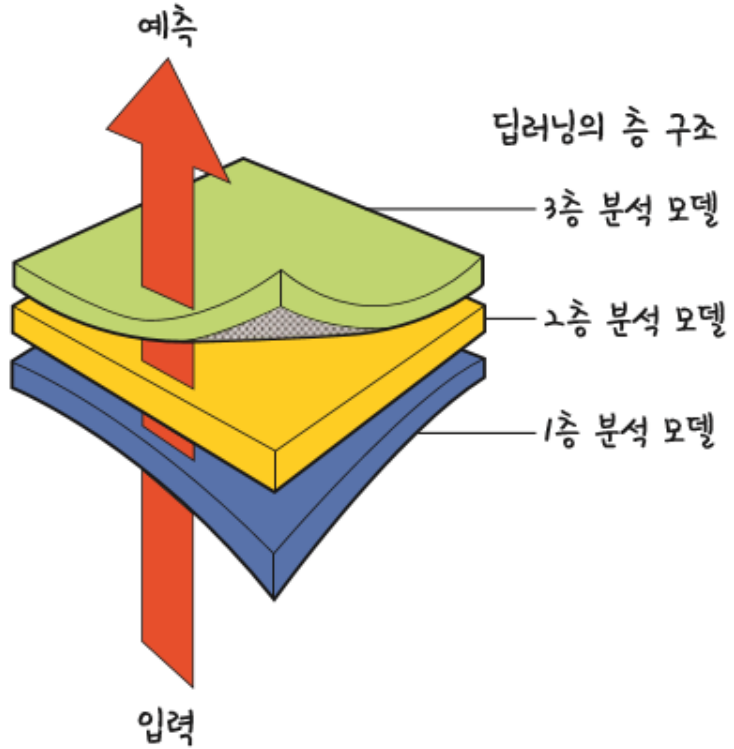
- 폴란드의 브로츠와프 의과대학에서 2013년 공개한 폐암 수술 환자의 수술 전 진단 데이터와 수술 후 생존 결과를 기록한 실제 의료 기록 데이터
- 470 라인, 라인 당 18개 항목

속성 / 클래스

항목 \ 줄	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	293	1	3.8	2.8	0	0	0	0	0	0	12	0	0	0	1	0	62	0
2	1	2	2.88	2.16	1	0	0	0	1	1	14	0	0	0	1	0	60	0
3	8	2	3.19	2.5	1	0	0	0	1	0	11	0	0	1	1	0	66	1 → 수술후 생존
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
470	447	8	5.2	4.1	0	0	0	0	0	0	12	0	0	0	0	0	49	0 → 수술후 사망

■ 모델 정의

- 딥러닝은 그림과 같이 여러 층이 쌓여 결과를 만들어 냄
- Sequential 함수는 딥러닝의 구조를 한 층 한 층 쉽게 쌓아올릴 수 있게 해 줌
- Sequential 함수를 선언할 때 필요한 층을 차례로 추가하면 됨



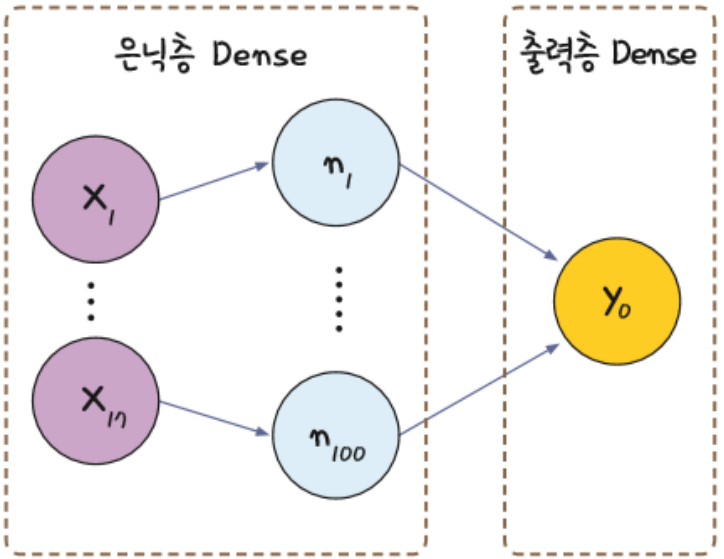
■ 모델 정의(코드)

- Keras 라이브러리 임포트

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
```

- 딥러닝 모델 정의

```
model = Sequential([
    Dense(30, input_shape=(17,), activation='relu'),
    Dense(1, activation='sigmoid')
])
```

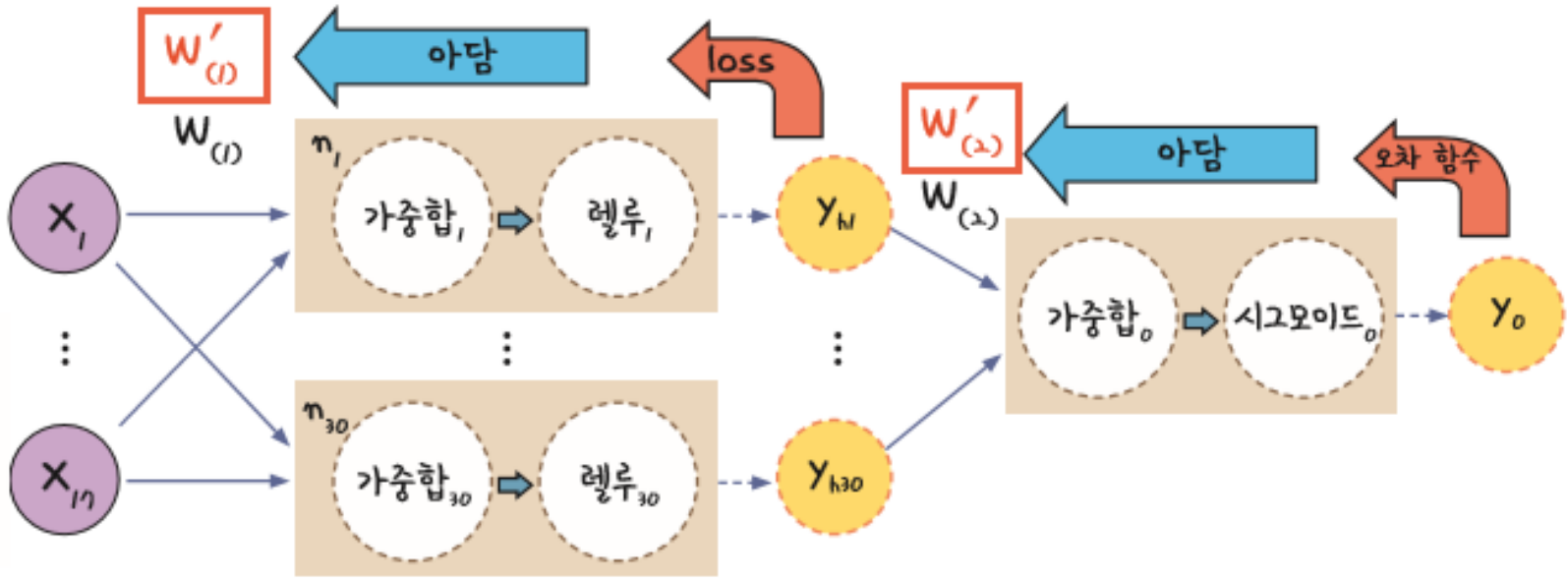


■ 모델 컴파일

- 지정한 모델이 효과적으로 구현될 수 있게 여러 가지 환경을 설정

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

- 오차 함수: binary_crossentropy
- 최적화 함수: 아담(adam)



■ 오차 함수

* 실제 값을 yt, 예측 값을 yo라고 가정할 때

평균 제곱 계열	mean_squared_error	평균 제곱 오차 계산: mean(square(yt − yo))
	mean_absolute_error	평균 절대 오차(실제 값과 예측 값 차이의 절댓값 평균) 계산: mean(abs(yt − yo))
	mean_absolute_percentage_error	평균 절대 백분율 오차(절댓값 오차를 절댓값으로 나눈 후 평균) 계산: mean(abs(yt − yo)/abs(yt) (단, 분모 ≠ 0)
	mean_squared_logarithmic_error	평균 제곱 로그 오차(실제 값과 예측 값에 로그를 적용한 값의 차이를 제곱한 값의 평균) 계산: mean(square((log(yo) + 1) − (log(yt) + 1)))
교차 엔트로피 계열	categorical_crossentropy	범주형 교차 엔트로피(일반적인 분류)
	binary_crossentropy	이항 교차 엔트로피(두 개의 클래스 중에서 예측할 때)

- 예측 값이 참과 거짓 둘 중 하나인 형식일 때는 binary_crossentropy(이항 교차 엔트로피)를 사용하는 것이 일반적

■ 모델 실행

- 모델 실행

```
model.fit(X, Y, epochs=20, batch_size=10)
```

- 에포크(epoch): 학습 프로세스가 모든 샘플에 대해 한 번 실행하는 것
- batch_size: 샘플을 한 번에 몇 개씩 처리할지를 정하는 부분
  - batch_size가 너무 크면 학습 속도가 느려지고,
    너무 작으면 각 실행 값의 편차가 생겨서 전체 결과값이 불안정해질 수 있음
  - 자신의 컴퓨터 메모리가 감당할 만큼의 batch_size를 찾아 설정

■ 피마 인디언 데이터 분석하기

- 비만은 유전일까? 아니면 식습관 조절에 실패한 자신의 탓일까?
- 비만이 유전 및 환경, 모두의 탓이라는 것을 증명하는 좋은 사례가 바로 미국 남서부에 살고 있는 피마 인디언의 사례
- 피마 인디언은 1950년대까지만 해도 비만인 사람이 단 한 명도 없는 민족이었음
- 그런데 지금은 전체 부족의 60%가 당뇨, 80%가 비만으로 고통받고 있음
- 이는 생존하기 위해 영양분을 체내에 저장하는 뛰어난 능력을 물려받은 인디언들이 미국의 기름진 패스트푸드 문화를 만나면서 벌어진 일

■ 피마 인디언 데이터 분석하기

- 768명의 인디언으로부터 8개의 정보와 1개의 클래스를 추출한 데이터

속성 / 클래스

	정보 1	정보 2	정보 3	…	정보 8	당뇨병 여부
1번째 인디언	6	148	72	…	50	1
2번째 인디언	1	85	66	…	31	0
3번째 인디언	8	183	64	…	32	1
…	…	…	…	…	…	…
768번째 인디언	1	93	70	…	23	0

샘플

- 속성: 8
  - 정보 1 (pregnant): 과거 임신 횟수
  - 정보 2 (plasma): 포도당 부하 검사 2시간 후 공복 혈당 농도(mm Hg)
  - 정보 3 (pressure): 확장기 혈압(mm Hg)
  - 정보 4 (thickness): 삼두근 피부 주름 두께(mm)
  - 정보 5 (insulin): 혈청 인슐린(2-hour, mu U/ml)
  - 정보 6 (BMI): 체질량 지수(BMI, weight in kg/(height in m)^2)
  - 정보 7 (pedigree): 당뇨병 가족력
  - 정보 8 (age): 나이
- 클래스: 당뇨(1), 당뇨 아님(0)

■ 피마 인디언 데이터 분석하기

❖ Pandas를 활용한 데이터 조사

```
import pandas as pd
df = pd.read_csv('dataset/pima-indians-diabetes.csv',
                 names = ["pregnant", "plasma", "pressure", "thickness",
                 "insulin", "BMI", "pedigree", "age", "class"])

print(df.head(5))
```

	pregnant	plasma	pressure	thickness	insulin	BMI	pedigree	age	class
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

■ 피마 인디언 데이터 분석하기

❖ Pandas를 활용한 데이터 조사

```
print(df.info())
```

Range Index: 768 entries, 0 to 767

Data	Columns (total 9)		
pregnant	768	non-null	int64
plasma	768	non-null	int64
pressure	768	non-null	int64
thickness	768	non-null	int64
insulin	768	non-null	int64
BMI	768	non-null	float64
pedigree	768	non-null	float64
age	768	non-null	int64
class	768	non-null	int64
Dtypes: float64(2) int64(7)			
Memory usage: 54.1 KB			

■ 피마 인디언 데이터 분석하기

❖ Pandas를 활용한 데이터 조사

```
print(df.describe())
```

	pregnant	plasma	pressure	thickness	insulin	BMI	pedigree	age	class
count	768	768	768	768	768	768	768	768	768
mean	3.845052	120.894531	69.105469	20.536458	79.799479	31.992578	0.471876	33.240885	0.348958
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.88416	0.331329	11.760232	0.476951
min	0	0	0	0	0	0	0.078	21	0
25%	1	99	62	0	0	27.3	0.24375	24	0
50%	3	117	72	23	30.5	32	0.3725	29	0
75%	6	140.25	80	32	127.25	36.6	0.62625	41	1
max	17	199	122	99	846	67.1	2.42	81	1

■ 피마 인디언 데이터 분석하기

❖ Pandas를 활용한 데이터 조사

- 임신 횟수와 당뇨병 발병 확률

```
print(df[['pregnant','class']].groupby(['pregnant'],
      as_index=False).mean().
      sort_values(by='pregnant', ascending=True))
```

	pregnant	class
0	0	0.342342
1	1	0.214815
2	2	0.184466
3	3	0.36
4	4	0.338235
5	5	0.368421
6	6	0.32
7	7	0.555556
8	8	0.578947
9	9	0.642857
10	10	0.416667
11	11	0.636364
12	12	0.444444
13	13	0.5
14	14	1
15	15	1
16	17	1

■ 피마 인디언 데이터 분석하기

❖ 그래프로 표현하기

- matplotlib, seaborn 패키지

```
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

- 그래프의 크기

```
plt.figure(figsize=(12,12))
```

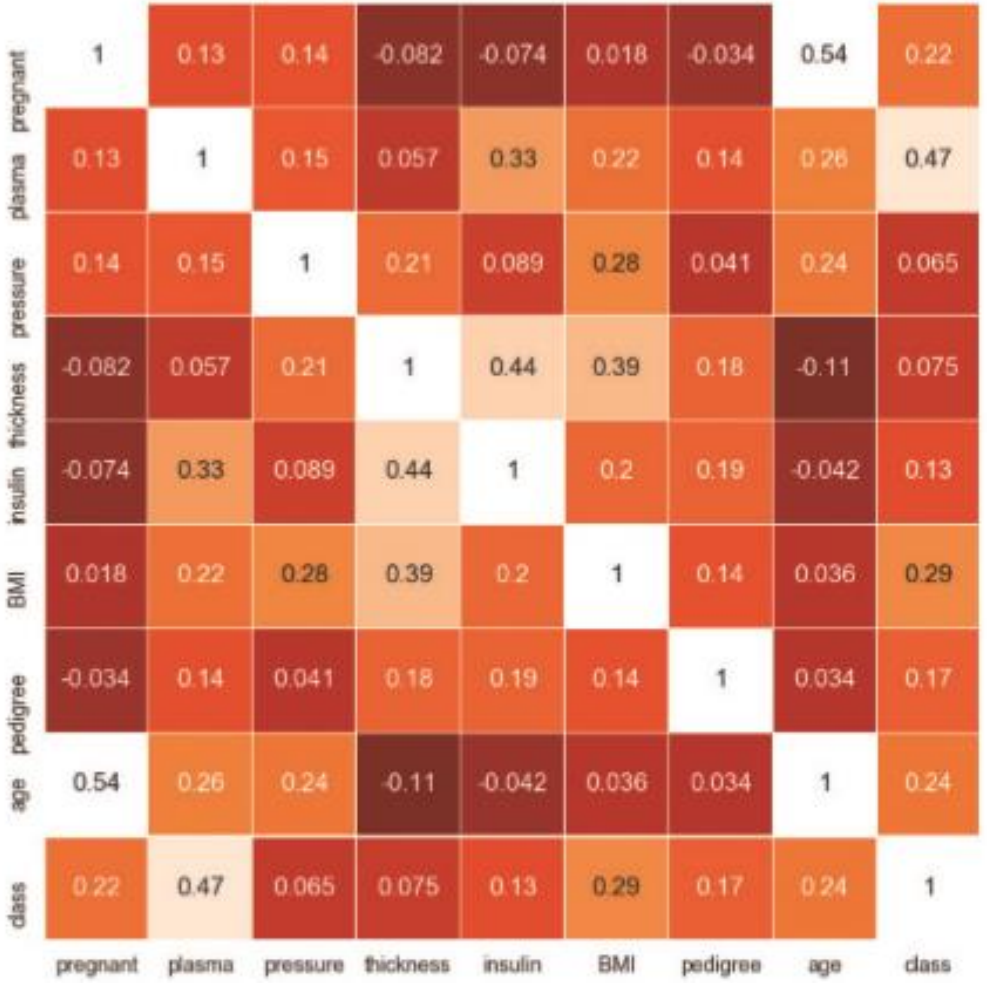
- seaborn 패키지의 항목간 상관관계를 보여주는 그래프 → heatmap

```
sns.heatmap(df.corr(), linewidths=0.1, vmax=0.5,
            cmap=plt.cm.gist_heat, linecolor='white', annot=True)
```

■ 피마 인디언 데이터 분석하기

❖ 그래프로 표현하기

- 두 항목이 전혀 다른 패턴으로 변화하고 있으면 0을, 서로 비슷한 패턴으로 변할수록 1에 가까운 값을 출력함
- plasma 항목(공복 혈당 농도)이 class 항목과 가장 상관관계가 높다는 것을 알 수 있음

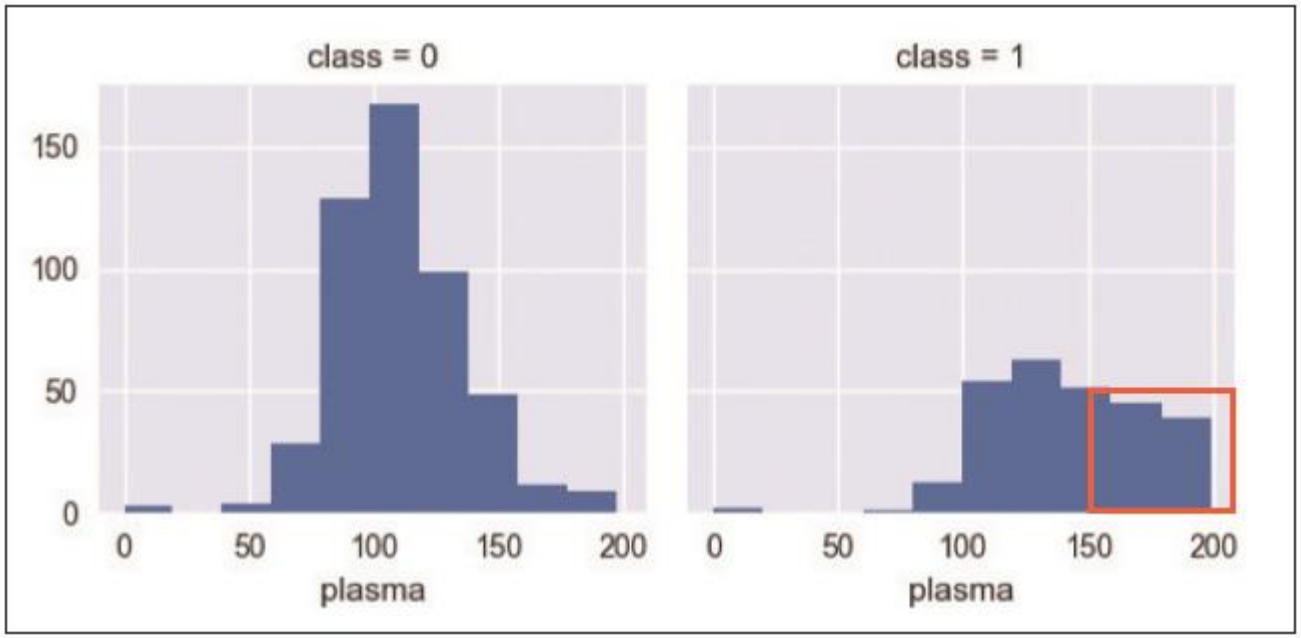


■ 피마 인디언 데이터 분석하기

❖ 그래프로 표현하기

- plasma와 class 항목 간의 관계 (히스토그램)

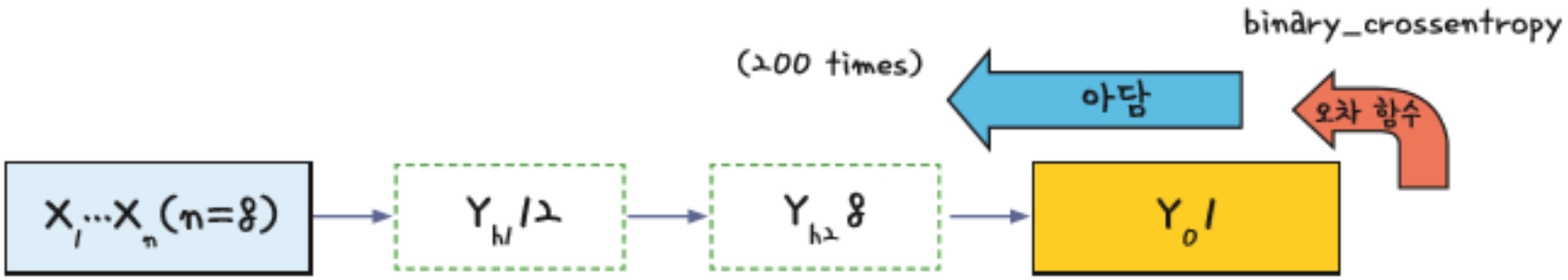
```
grid = sns.FacetGrid(df, col='class')
grid.map(plt.hist, 'plasma', bins=10)
plt.show()
```



■ 피마 인디언의 당뇨병 예측 실행

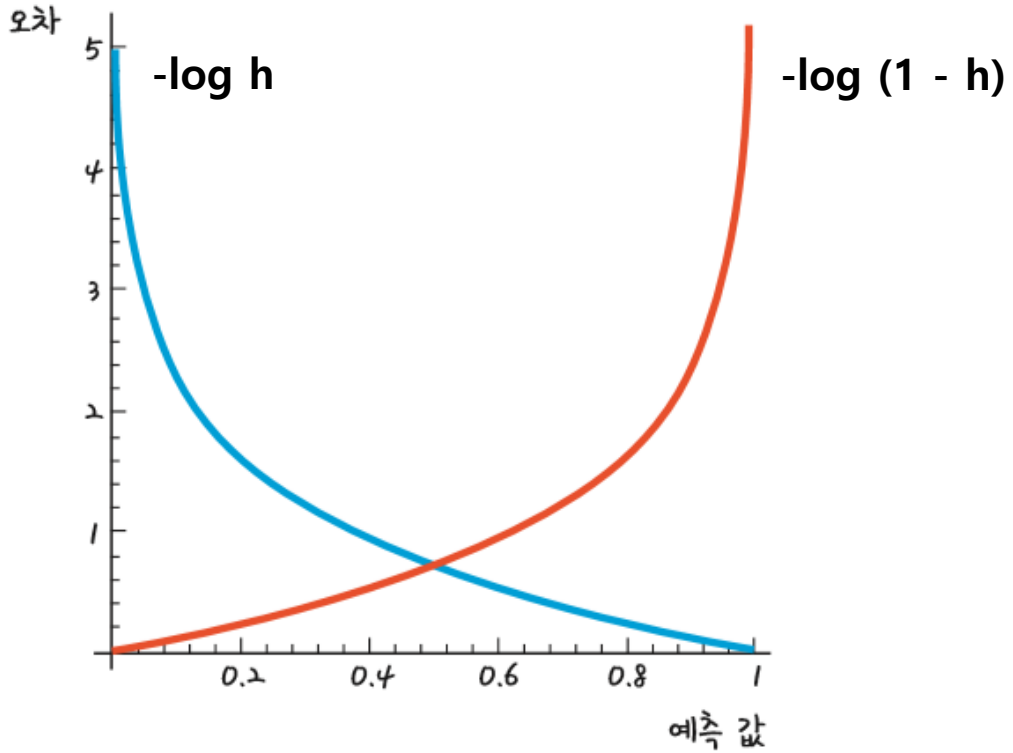
- 모델 설계

```
model = Sequential([
    Dense(12, input_shape=(8,), activation='relu'),
    Dense(8, activation='relu'),
    Dense(1, activation='sigmoid')
])
```



■ Binary Crossentropy

❖ 오차



- y의 실제 값이 1일 때 –log h 그래프를 쓰고, 0일 때 –log (1 - h) 그래프를 써야 함

$$-\{\underbrace{y\log h}_{A}+\underbrace{(1-y)\log(1-h)}_{B}\}$$

■ 아이리스 품종 예측


Iris-virginica


Iris-setosa


Iris-versicolor

- 아이리스는 그 꽃봉오리가 마치 먹물을 머금은 붓과 같다 하여 우리나라에서 '붓꽃' 이라고 불리우는 아름다운 꽃
- 아이리스는 꽃잎의 모양과 길이에 따라 여러 가지 품종으로 나뉨

■ 아이리스 품종 데이터

속성 / 클래스

	정보 1	정보 2	정보 3	정보 4	품종
1번째 아이리스	5.1	3.5	4.0	0.2	Iris-setosa
2번째 아이리스	4.9	3.0	1.4	0.2	Iris-setosa
3번째 아이리스	4.7	3.2	1.3	0.3	Iris-setosa
...	...	...	...	...	...
150번째 아이리스	5.9	3.0	5.1	1.8	Iris-virginica

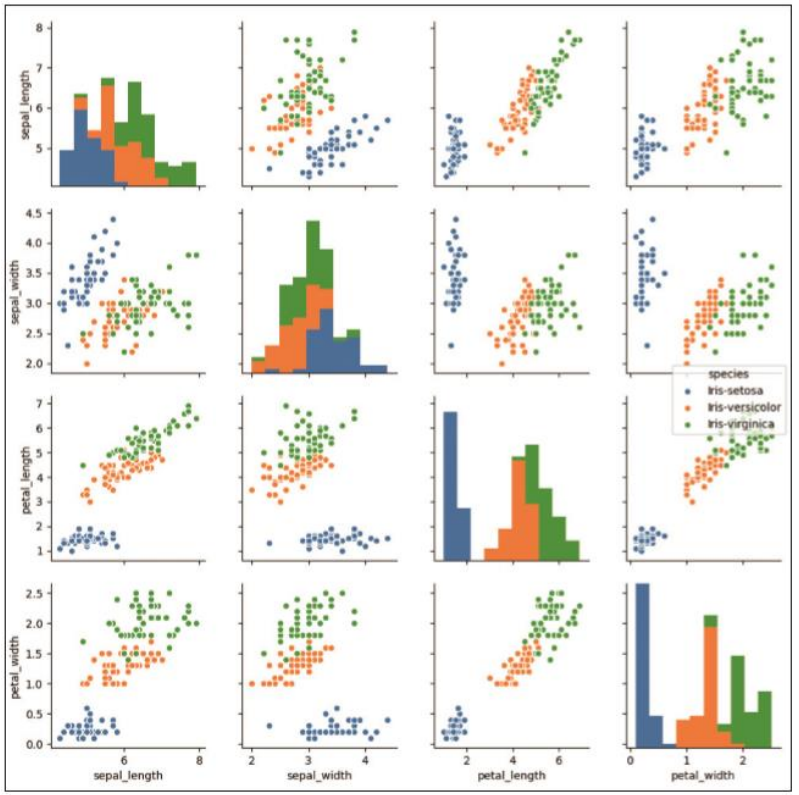
샘플

- 샘플 수: 150
- 속성 수: 4
  - 정보 1: 꽃받침 길이 (sepal length, 단위: cm)
  - 정보 2: 꽃받침 넓이 (sepal width, 단위: cm)
  - 정보 3: 꽃잎 길이 (petal length, 단위: cm)
  - 정보 4: 꽃잎 넓이 (petal width, 단위: cm)
- 클래스: Iris-setosa, Iris-versicolor, Iris-virginica

■ 아이리스 품종 데이터

❖ 상관도 그래프

```
import pandas as pd
df = pd.read_csv('../dataset/iris.csv', columns = ["sepal_length", "sepal_width",
                    "petal_length", "petal_width", "species"])

import seaborn as sns
import matplotlib.pyplot as plt

sns.pairplot(df, hue='species');
plt.show()
```



■ One-hot Encoding

- Iris 품종 데이터

```
df = pd.read_csv('dataset/iris.csv', names = ["sepal_length",
                "sepal_width", "petal_length", "petal_width", "species"])

dataset = df.values
X = dataset[:,0:4].astype(float)
Y_obj = dataset[:,4]
```

- array(['Iris-setosa', 'Iris-versicolor', 'Iris-virginica']) → array([0,1,2]) 변환

```
from sklearn.preprocessing import LabelEncoder

e = LabelEncoder()
e.fit(Y_obj)
Y = e.transform(Y_obj)
```

■ One-hot Encoding

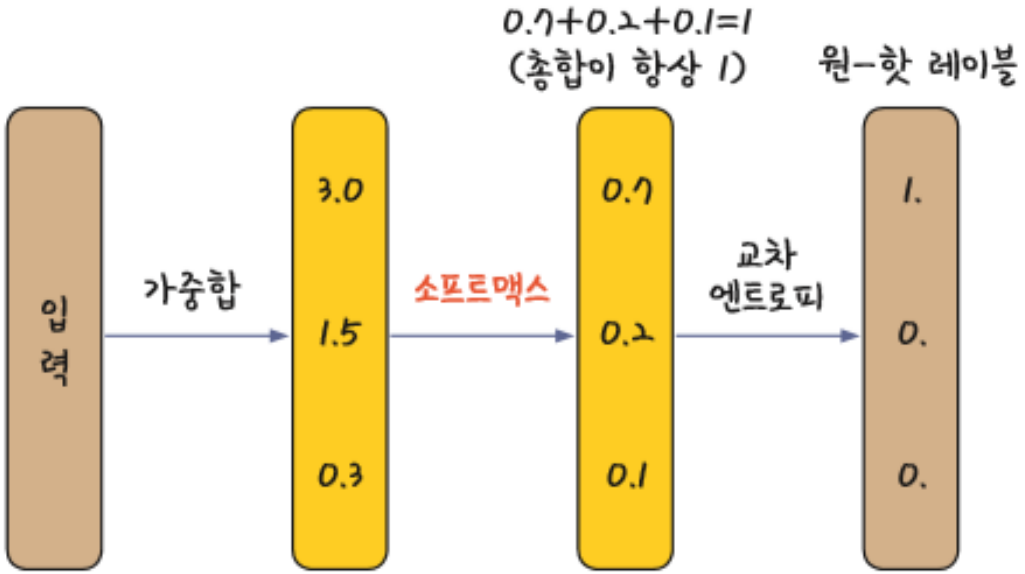
- 활성화 함수를 적용하려면 Y 값이 0과 1로 되어야만 함

```
from tensorflow.keras.utils import to_categorical
Y_encoded = to_categorical(Y)
```

- array([0,1,2]) → array([[1., 0., 0.], [0., 1., 0.], [0., 0., 1.]]) 로 변환
- 여러 개의 Y 값을 0과 1로만 이루어진 형태로 바꿔주는 기법을 원-핫 인코딩(one-hot-encoding)이라고 함

■ 소프트맥스(Softmax)

- 총합이 1인 형태로 바꿔서 계산해 주는 함수



- 합계가 1인 형태로 변환하면 큰 값이 두드러지게 나타나고 작은 값은 더 작아짐
- 이 값이 교차 엔트로피(categorical cross entropy)를 지나 [1., 0., 0.]으로 변화하게 되면 우리가 원하는 원-핫 인코딩 값, 즉 하나만 1이고 나머지는 모두 0인 형태로 전환시킬 수 있음

■ 초음파 광물 데이터

❖ 실험 배경

- 1988년 존스홉킨스대학교의 세즈노프스키(Sejnowski) 교수는 2년 전 힌튼 교수가 발표한 역전파 알고리즘에 관심을 가지고 있었음
- 그는 은닉층과 역전파가 얼마나 큰 효과가 있는지를 직접 실험해 보기 위해 광석과 일반 돌을 갖다 놓고 음파 탐지기를 쏜 후 그 결과를 데이터로 정리
- 오차 역전파 알고리즘을 사용한 신경망이 과연 얼마나 광석과 돌을 구분하는데 효과적인지 알아보기 위함

■ 초음파 광물 데이터

❖ 데이터 확인

```
import pandas as pd
df = pd.read_csv('dataset/sonar.csv', header=None)
df.head()
```

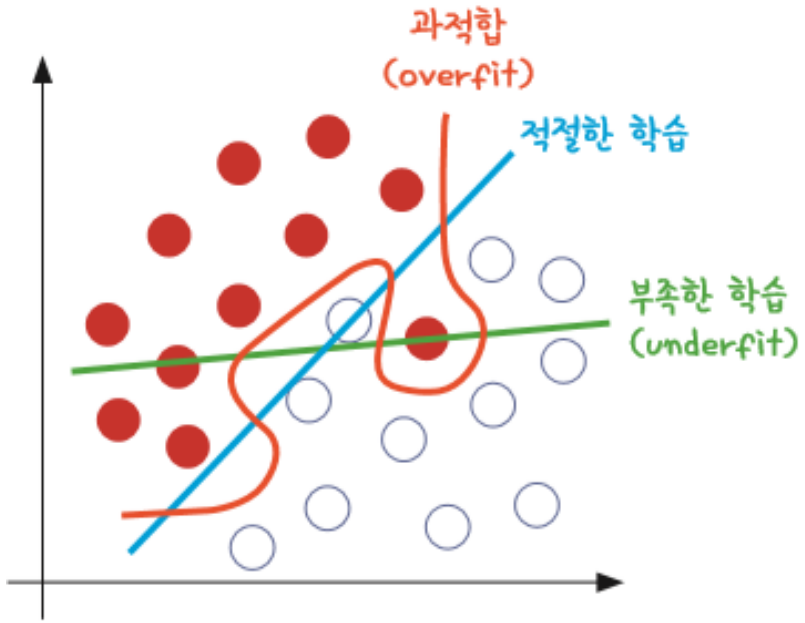
	0	1	2	3	...	59	60
0	0.02	0.0371	0.0428	0.0207	...	0.0032	R
1	0.0453	0.0523	0.0843	0.0689	...	0.0044	R
2	0.0262	0.0582	0.1099	0.1083	...	0.0078	R
3	0.01	0.0171	0.0623	0.0205	...	0.0117	R
4	0.0762	0.0666	0.0481	0.0394	...	0.0094	R

- 총 샘플의 수는 208개이고, 60개의 속성과 1개의 클래스로 이루어져 있음

■ 과적합 이해하기

❖ 과적합(overfitting)이란?

- 모델이 학습 데이터셋 안에서는 일정 수준 이상의 예측 정확도를 보이지만, 새로운 데이터에 적용하면 잘 맞지 않는 것을 말함



■ 과적합 이해하기

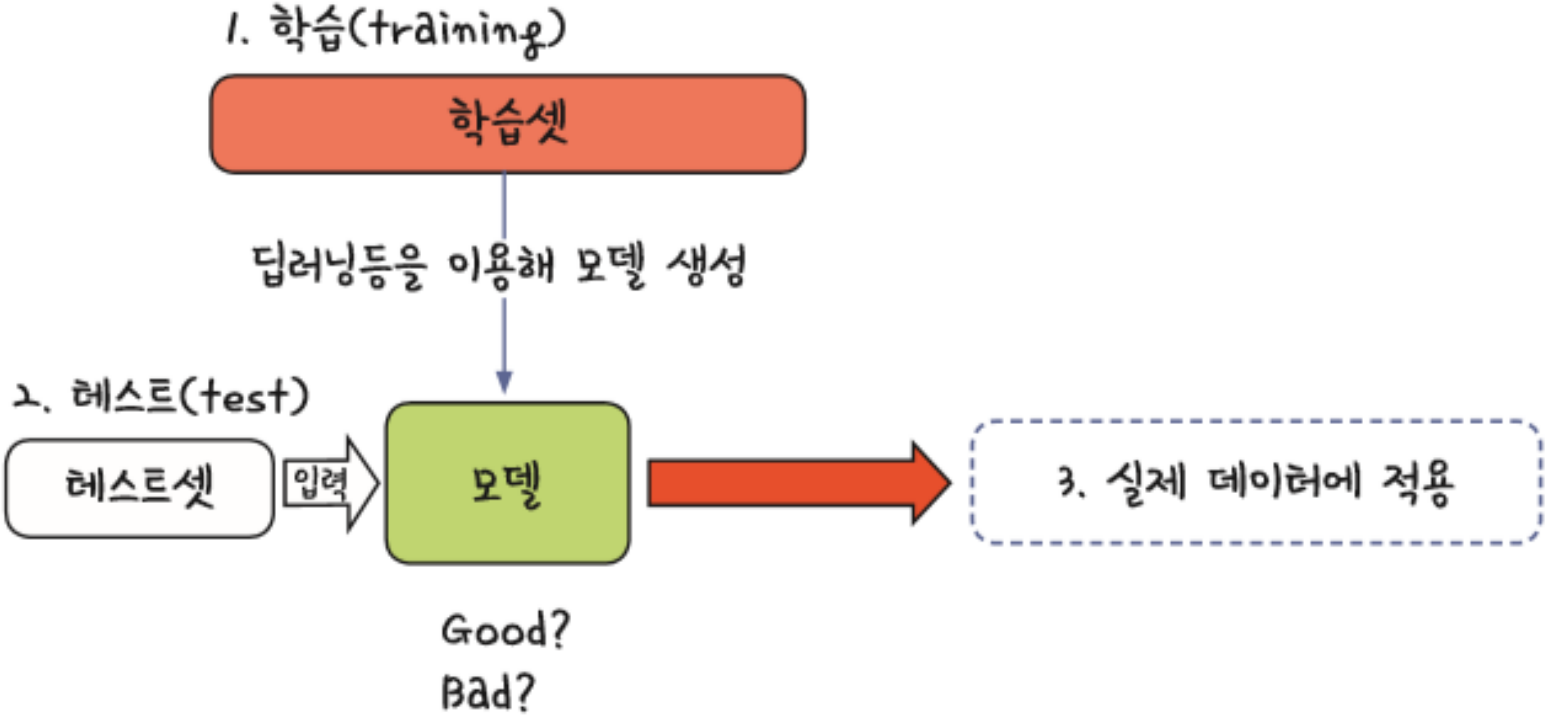
❖ 방지 방법

- 학습을 하는 데이터셋과 이를 테스트할 데이터셋을 완전히 구분한 다음 학습과 동시에 테스트를 병행하며 진행하는 것이 한 방법
- 예를 들어, 데이터셋이 총 100개의 샘플로 이루어져 있다면 다음과 같이 두 개의 셋으로 나눔

70개 샘플은 학습셋으로	30개 샘플은 테스트셋으로

- 신경망을 만들어 70개의 샘플로 학습을 진행한 후 이 학습의 결과를 저장
  → 이렇게 저장된 파일을 '모델'이라고 부름
- 모델은 다른 셋에 적용할 경우 학습 단계에서 각인되었던 그대로 다시 수행함
- 따라서 나머지 30개의 샘플로 실험해서 정확도를 살펴보면 학습이 얼마나 잘 되었는지를 알 수 있음

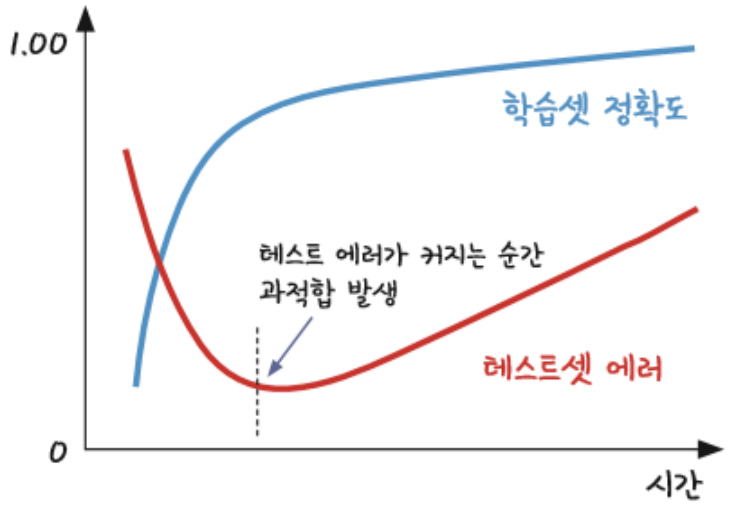
■ 학습셋과 테스트셋



- 학습셋만 가지고 평가할 때, 층을 더하거나 에포크(epoch) 값을 높여 실행 횟수를 늘리면 정확도가 계속해서 올라갈 수 있음
- 하지만 학습 데이터셋만으로 평가한 예측 성공률이 테스트셋에서도 그대로 나타나지는 않음

■ 학습셋과 테스트셋

- 학습이 깊어져서 학습셋 내부에서의 성공률은 높아져도 테스트셋에서는 효과가 없다면 과적합이 일어나고 있는 것



- 학습을 진행해도 테스트 결과가 더 이상 좋아지지 않는 지점에서 학습을 멈춰야 함
- 이때의 학습 정도가 가장 적절한 것으로 볼 수 있음

■ 학습셋과 테스트셋

❖ 세즈노프스키 교수 실험 결과

- 은닉층(Number of Hidden Units) 수가 올라감에 따라 학습셋의 예측률(Average Performance on Training Sets)과 테스트셋의 예측률(Average Performance on Testing Sets)의 변화

은닉층 수의 변화	학습셋의 예측률	테스트셋의 예측률
0	79.3	73.1
2	96.2	85.7
3	98.1	87.6
6	99.4	89.3
12	99.8	90.4
24	100	89.2

■ 학습셋과 테스트셋

- 학습셋과 테스트셋으로 분리하는 코드

```
from sklearn.model_selection import train_test_split

# 학습셋과 테스트셋의 구분
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.3,
                                                    random_state=seed)
```

- 모델 실행 코드

```
model.fit(X_train, Y_train, epochs=130, batch_size=5)

# 테스트셋에 모델 적용
print("\n Test Accuracy: %.4f" % (model.evaluate(X_test, Y_test) [1]))
```

■ 모델 저장과 재사용

- 학습이 끝난 후 테스트해 본 결과가 만족스러울 때 이를 모델로 저장하여 새로운 데이터에 사용할 수 있음
- 학습한 결과를 저장

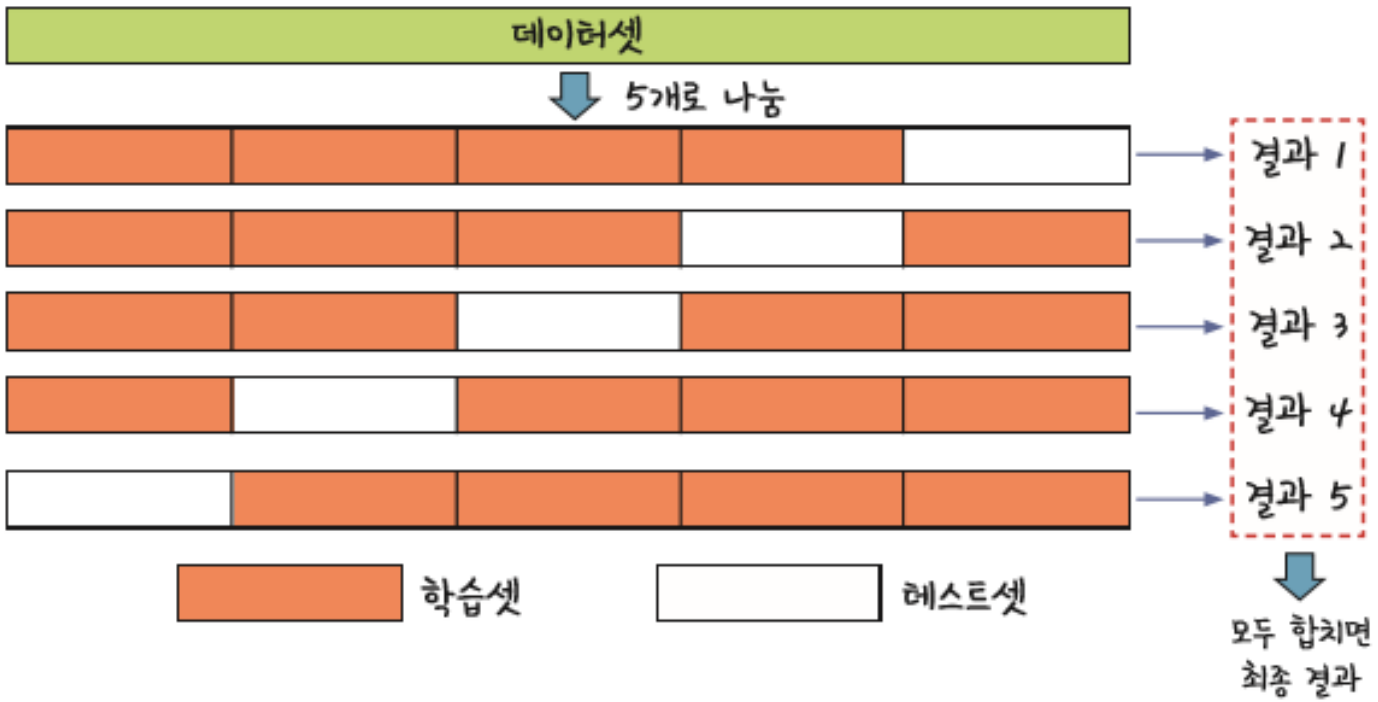
```
from tensorflow.keras.models import load_model

model.save('my_model.h5')
```

- 불러오는 방법

```
model = load_model ('my_model.h5')
```

■ k겹 교차 검증

- k겹 교차 검증이란 데이터셋을 여러 개로 나누어 하나씩 테스트셋으로 사용하고 나머지를 모두 합해서 학습셋으로 사용하는 방법
- 이렇게 하면 가지고 있는 데이터의 100%를 테스트셋으로 사용할 수 있음
- 예를 들어, 5겹 교차 검증(5-fold cross validation)의 예



■ 데이터 확인

❖ 와인 측정 데이터

- 포르투갈 서북쪽의 대서양을 맞닿고 위치한 비뉴 베르드(Vinho Verde) 지방에서 만들어진 와인을 측정한 데이터
- 레드와인 샘플 1,599개를 등급과 맛, 산도를 측정해 분석하고 화이트와인 샘플 4,898개를 마찬가지로 분석해 데이터를 만들었음

- 데이터 샘플링 (비율을 인수로 제공)

```
df_pre = pd.read_csv('dataset/wine.csv', header=None)
df = df_pre.sample(frac=1)
```

■ 데이터 확인

❖ 와인 측정 데이터

- 총 6497개의 샘플
- 13개의 속성

0	주석산 농도	7	밀도
1	아세트산 농도	8	pH
2	구연산 농도	9	황산칼륨 농도
3	잔류 당분 농도	10	알코올 도수
4	염화나트륨 농도	11	와인의 맛(0~10등급)
5	유리 아황산 농도	12	class (1: 레드와인, 0: 화이트와인)
6	총 아황산 농도		

■ 와인 종류 예측 코딩

❖ 딥러닝 모델

- 4개의 은닉층을 만들어 각각 30, 12, 8, 1개의 노드를 줌
- 이항 분류(binary classification) 문제

  → 오차 함수는 binary_crossentropy, 최적화 함수로 adam
- 전체 샘플이 200회 반복되어 입력될 때까지 실험을 반복 (epochs=200)
- 한 번에 입력되는 입력 값은 200개씩 (batch_size=200)

■ 모델 업데이트

❖ 에포크(epoch)마다 모델의 정확도를 기록하면서 저장

- 모델이 저장될 폴더를 지정
- 에포크 횟수와 이때의 테스트셋 오차 값을 이용해 파일 이름을 만들어 hdf5라는 확장자로 저장
- 예를 들어, 100번째 에포크를 실행하고 난 결과 오차가 0.0612라면, 파일명은 100-0.0612.hdf5가 되는 것

```
import os

MODEL_DIR = './models/'
if not os.path.exists(MODEL_DIR):
    os.mkdir(MODEL_DIR)

modelpath="./models/{epoch:03d}-{val_loss:.4f}.hdf5"
```

■ 모델 업데이트

❖ 모니터링

- 콜백(callback) 함수

```
from keras.callbacks import ModelCheckpoint
checkpointer = ModelCheckpoint(filepath=modelpath, monitor='val_loss', verbose=1)

model.fit(X, Y, validation_split=0.2, epochs=200, batch_size=200,
          verbose=0, callbacks=[checkpointer])
```

```
Epoch 00194: saving model to ./model/194-0.0629.hdf5
Epoch 00195: saving model to ./model/195-0.0636.hdf5
Epoch 00196: saving model to ./model/196-0.0630.hdf5
Epoch 00197: saving model to ./model/197-0.0695.hdf5
Epoch 00198: saving model to ./model/198-0.0724.hdf5
Epoch 00199: saving model to ./model/199-0.0635.hdf5
```

- 베스트 모델만 저장

```
checkpointer = ModelCheckpoint(filepath=modelpath, monitor='val_loss',
                               verbose=1, save_best_only=True)
```

■ 그래프로 확인

❖ 에포크에 따른 정확도와 테스트 결과

- 에포크를 얼마나 지정할지를 결정하는 것이 중요

  → 학습을 반복하는 횟수가 너무 적어도 안 되고 또 너무 많아도 과적합을 일으키므로 문제가 있음

- 모델이 학습되는 과정을 history 변수를 만들어 저장

```
df = df_pre.sample(frac=0.15)

history = model.fit(X, Y, validation_split=.33, epochs=2000, batch_size=500)
```

■ 그래프로 확인

- 학습셋

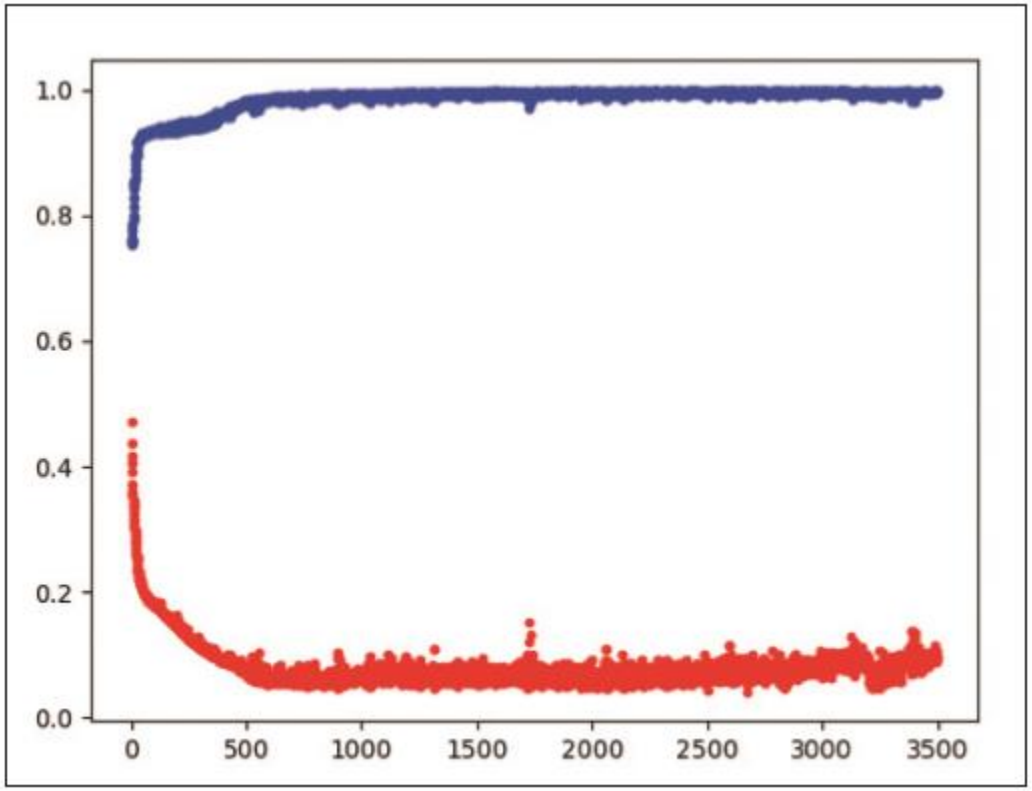
  → 정확도('accuracy'), 오차값('loss')

- 테스트셋

  → 정확도('val_accuracy'), 오차값('val_loss')

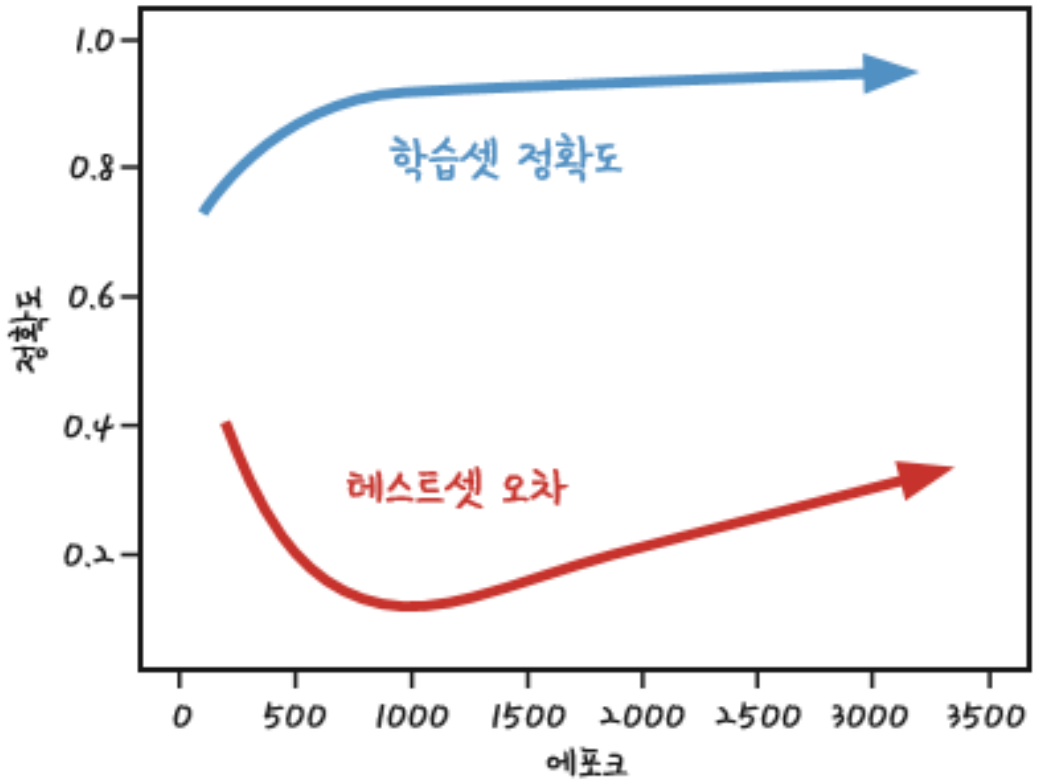
```
import matplotlib.pyplot as plt
%matplotlib inline

y_vloss=history.history['val_loss']
y_acc=history.history['accuracy']

x_len = numpy.arange(len(y_acc))
plt.plot(x_len, y_vloss, "o", c="red", markersize=3)
plt.plot(x_len, y_acc, "o", c="blue", markersize=3)
```

■ 그래프로 확인하는 코드



■ 그래프로 확인하는 코드

- 학습이 진행될수록 학습셋의 정확도는 오르지만 테스트셋에서는 과적합 발생



■ 학습의 자동 중단

- 학습이 진행될수록 학습셋의 정확도는 올라가지만 과적합으로 인해 테스트셋의 실험 결과는 점점 나빠지게 됨
- 학습이 진행되어도 테스트셋 오차가 줄지 않으면 학습을 멈추게 하는 함수

  → EarlyStopping() 함수

```
early_stopping_callback = EarlyStopping(monitor='val_loss', patience=100)

model.fit(X, Y, validation_split=0.2, epochs=3500, batch_size=500, verbose=0,
          callbacks=[early_stopping_callback, checkpointer])
```

■ 보스턴 집값 예측

❖ 데이터 출처

- 1978년, 집값에 가장 큰 영향을 미치는 것이 '깨끗한 공기'라는 연구 결과가 하버드 대학교 도시개발학과에서 발표됨
- 이들은 자신의 주장을 뒷받침하기 위해 집값의 변동에 영향을 미치는 여러 가지 요인을 모아서 환경과 집값의 변동을 보여주는 데이터셋을 만듦
- 이것이 현재 선형 회귀를 테스트하는 가장 유명한 데이터로 쓰이고 있음

❖ 선형회귀 문제

- 하나의 정답을 맞히는 것이 아니라 수치를 예측하는 문제

■ 보스턴 집값 예측

❖ 데이터 확인

- 총 샘플의 수는 506개
- 13개의 속성과 1개의 클래스

0	CRIM: 인구 1인당 범죄 발생 수
1	ZN: 25,000평방 피트 이상의 주거 구역 비중
2	INDUS: 소매업 외 상업이 차지하는 면적 비율
3	CHAS: 찰스강 위치 변수(1: 강 주변, 0: 이외)
4	NOX: 일산화질소 농도
5	RM: 집의 평균 방 수
6	AGE: 1940년 이전에 지어진 비율
7	DIS: 5가지 보스턴 시 고용 시설까지의 거리
8	RAD: 순환고속도로의 접근 용이성
9	TAX: $10,000당 부동산 세율 총계
10	PTRATIO: 지역별 학생과 교사 비율
11	B: 지역별 흑인 비율
12	LSTAT: 급여가 낮은 직업에 종사하는 인구 비율(%)
13	가격(단위: $1,000)

■ 보스턴 집값 예측

❖ 선형 회귀 실행

- 선형 회귀 데이터는 마지막에 참과 거짓을 구분할 필요가 없음
- 출력층에 활성화 함수를 지정할 필요도 없음

```
model = Sequential([
    Dense(30, input_dim=13, activation='relu'),
    Dense(6, activation='relu'),
    Dense(1)
])
```

- 모델의 학습이 어느 정도 되었는지 확인하기 위해 예측 값과 실제 값을 비교하는 부분을 추가

```
Y_prediction = model.predict(X_test).flatten()
for i in range(10):
    label = Y_test[i]
    prediction = Y_prediction[i]
    print("실제가격: {:.3f}, 예상가격: {:.3f}".format(label, prediction))
```