과적합 피하기

[실습] 초음파 광물 예측

1 | 데이터의 확인과 실행

2 | 과적합 이해하기

3 | 학습셋과 테스트셋

4 | 모델 저장과 재사용

5 | k겹 교차 검증

- **실습 데이터** 초음파 광물 예측 : dataset/sonar.csv

- 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 구분하는 데 효과적인지 직접 알아보자

1 | 데이터의 확인과 실행

- 먼저 데이터를 확인해 보자
- Dataset/sonar.csv 파일을 가져옴

```
import pandas as pd

df = pd.read_csv('../dataset/sonar.csv', header=None)
print(df.info())
```

1 | 데이터의 확인과 실행

Range Index: 208 entries,0 to 207			
Data columns (total 61 columns):			
0	208	non-null	float64
1	208	non-null	float64
2	208	non-null	float64
3	208	non-null	float64
4	208	non-null	float64
5	208	non-null	float64
...	...	...	...
58	208	non-null	float64
59	208	non-null	float64
60	208	non-null	object
Dtypes: float64(60), object(1)			
memory usage: 99.2+ KB			

1 | 데이터의 확인과 실행

- Index가 208개이므로 총 샘플의 수는 208개이고, 컬럼 수가 61개이므로 60개의 속성과 1개의 클래스로 이루어져 있음을 짐작할 수 있음
- 모든 컬럼이 실수형(float64)인데, 맨 마지막 컬럼만 객체형인 것으로 보아 마지막에 나오는 컬럼은 클래스이며 데이터형 변환이 필요한 것을 알 수 있음
- 실제로 맞는지 일부를 출력해 확인해 보자

```
print(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

1 | 데이터의 확인과 실행

- 위의 정보를 토대로 이제 다음과 같이 딥러닝을 실행해 보자

코드 13-1 초음파 광물 예측하기: 데이터 확인과 실행

- **예제 소스** deep_code/04-Sonar.py

```
from keras.models import Sequential
from keras.layers.core import Dense
from sklearn.preprocessing import LabelEncoder

import pandas as pd
import numpy
import tensorflow as tf

# seed 값 설정
seed = 0
numpy.random.seed(seed)
tf.set_random_seed(seed)
```

1 | 데이터의 확인과 실행

```
# 데이터 입력
df = pd.read_csv('../dataset/sonar.csv', header=None)

dataset = df.values
X = dataset[:,0:60]
Y_obj = dataset[:,60]

# 문자열 변환
e = LabelEncoder()
e.fit(Y_obj)
Y = e.transform(Y_obj)

# 모델 설정
model = Sequential()
model.add(Dense(24, input_dim=60, activation='relu'))
model.add(Dense(10, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

1 | 데이터의 확인과 실행

```
# 모델 컴파일
model.compile(loss='mean_squared_error',
              optimizer='adam',
              metrics=['accuracy'])

# 모델 실행
model.fit(X, Y, epochs=200, batch_size=5)

# 결과 출력
print("\n Accuracy: %.4f" % (model.evaluate(X, Y)[1]))
```

1 | 데이터의 확인과 실행

- 실행 결과

```
Epoch 1/200
208/208 [==============================] - 0s - loss: 0.2532 - acc: 0.4567
Epoch 2/200
208/208 [==============================] - 0s - loss: 0.2457 - acc: 0.5385
Epoch 3/200
208/208 [==============================] - 0s - loss: 0.2426 - acc: 0.5385

(중략)

Epoch 198/200
208/208 [==============================] - 0s - loss: 0.0057 - acc: 0.9952
Epoch 199/200
208/208 [==============================] - 0s - loss: 0.0057 - acc: 0.9952
Epoch 200/200
208/208 [==============================] - 0s - loss: 0.0057 - acc: 0.9952
 32/208 [===>..........................] - ETA: 0s
 Accuracy: 0.9952
```

- 이 모델의 정확도는 99.52%

2 | 과적합 이해하기

- **과적합**(over fitting)이란 모델이 학습 데이터셋 안에서는 일정 수준 이상의 예측 정확도를 보이지만, 새로운 데이터에 적용하면 잘 맞지 않는 것을 말함

- 그림 13-1의 그래프에서 빨간색을 보면 주어진 샘플에 정확히 맞게끔 선이 그어져 있음
- 하지만 이 선은 너무 이 경우에만 최적화되어 있음
- 다시 말해서, 완전히 새로운 데이터에 적용하면 이 선을 통해 정확히 두 그룹을 나누지 못하게 된다는 의미

2 | 과적합 이해하기

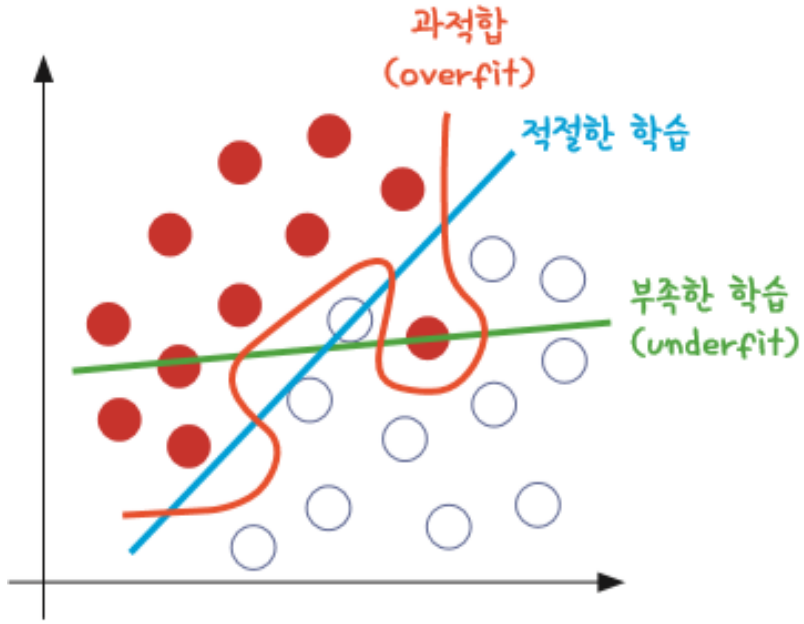

그림 13-1 과적합이 일어난 경우 (빨간색)와 학습이 제대로 이루어지지 않은 경우(초록색)

2 | 과적합 이해하기

- 과적합은 층이 너무 많거나 변수가 복잡해서 발생하기도 하고 테스트셋과 학습셋이 중복될 때 생기기도 함
- 특히 딥러닝은 학습 단계에서 입력, 은닉층, 출력층의 노드들에 상당히 많은 변수들이 투입됨

  → 따라서 딥러닝을 진행하는 동안 과적합에 빠지지 않게 늘 주의해야 한다!

3 | 학습셋과 테스트셋

- 그렇다면 과적합을 방지하려면 어떻게 해야 할까?
- 먼저 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

3 | 학습셋과 테스트셋

- 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 있는 것

- 딥러닝 같은 알고리즘을 충분히 조절하여 가장 나은 모델이 만들어지면, 이를 실생활에 대입하여 활용하는 것이 바로 머신러닝의 개발 순서

3 | 학습셋과 테스트셋

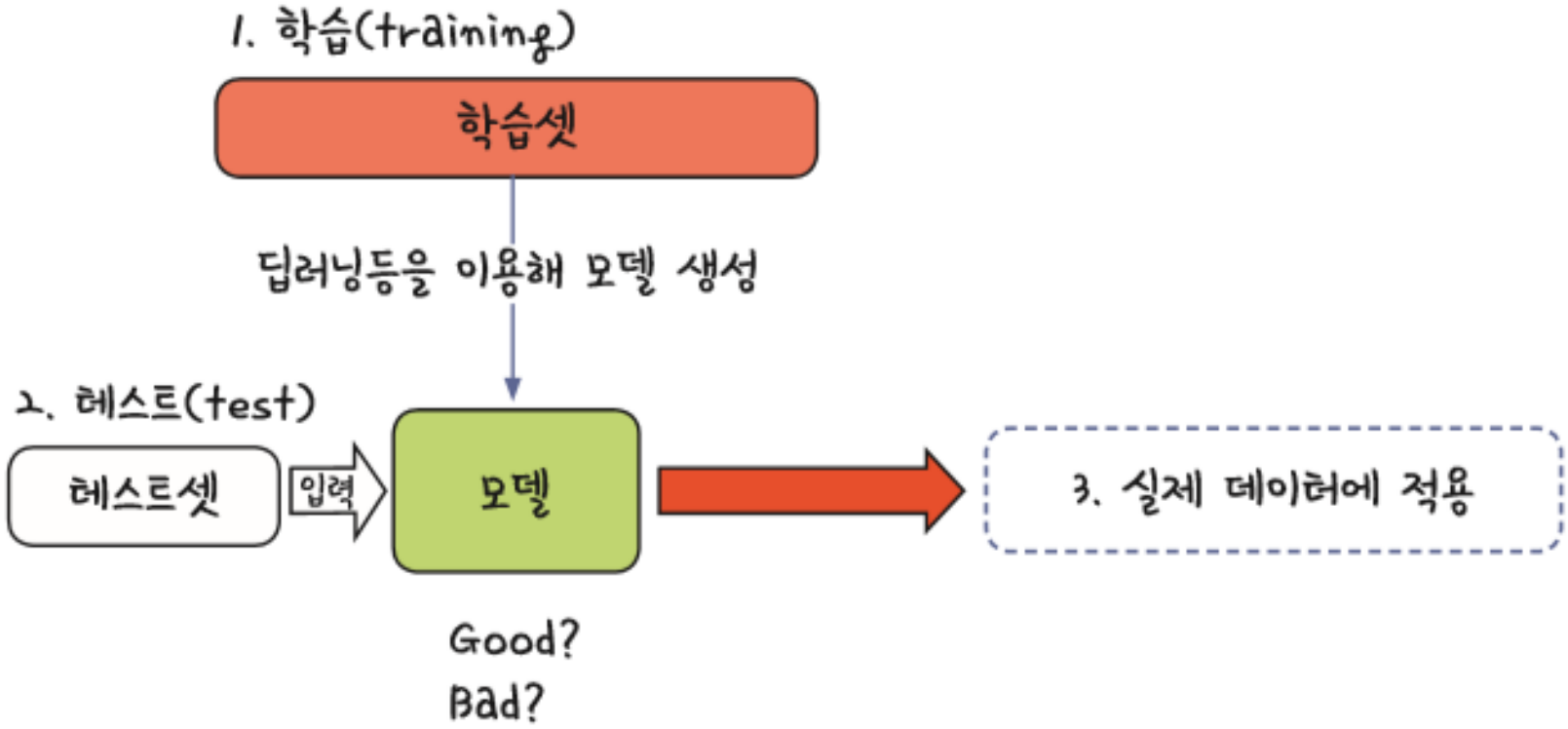

그림 13-2 학습셋과 테스트셋

3 | 학습셋과 테스트셋

- [과적합의 발생] : 학습셋만 가지고 평가할때, 층을 더하거나 에포크(epoch) 값을 높여 실행 횟수를 늘리면 정확도가 계속해서 올라갈 수 있음
- 하지만 학습 데이터셋만으로 평가한 예측 성공률이 테스트셋에서도 그대로 나타나지는 않음
- 학습이 깊어져서 학습셋 내부에서의 성공률은 높아져도 테스트셋에서는 효과가 없다면 과적합이 일어나고 있는 것

3 | 학습셋과 테스트셋

- 이를 그래프로 표현하면 그림 13-3과 같음

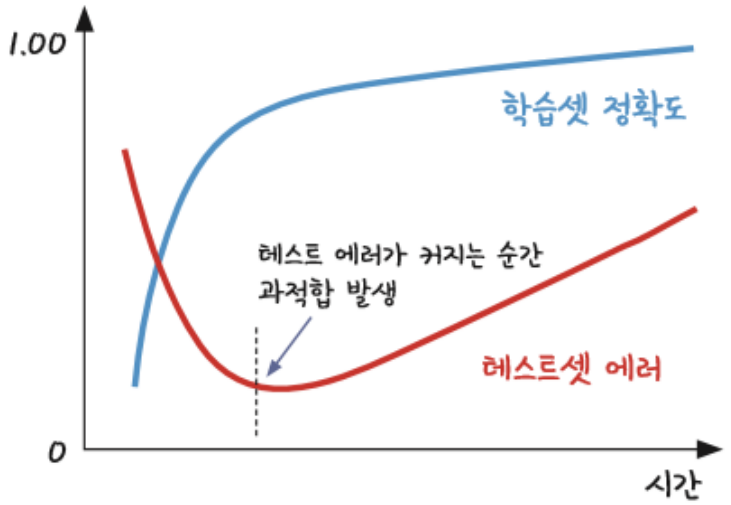

그림 13-3 학습이 계속되면 학습셋에서의 정확도는 계속 올라가지만, 테스트셋에서는 과적합이 발생!

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

3 | 학습셋과 테스트셋

- 우리가 다루는 초음파 광물 예측 데이터를 만든 세즈노프스키 교수가 실험 결과를 발표한 논문의 일부를 가져와 보자

TABLE 2
Aspect-Angle Dependent Series

Number of Hidden Units	Average Performance on Training Sets (%)	Standard Deviation on Training Sets (%)	Average Performance on Testing Sets (%)	Standard Deviation on Testing Sets (%)
0	79.3	3.4	73.1	4.8
2	96.2	2.2	85.7	6.3
3	98.1	1.5	87.6	3.0
6	99.4	0.9	89.3	2.4
12	99.8	0.6	90.4	1.8
24	100.0	0.0	89.2	1.4

Summary of the results of the aspect-angle dependent series of experiments with training and testing sets selected to include all target aspect angles. The standard deviation shown is across networks with different initial conditions.

그림 13-4 학습셋과 테스트셋 정확도 측정의 예(RP Gorman et.al., 1998)

3 | 학습셋과 테스트셋

- 여기서 눈여겨 봐야 할 부분은 은닉층(Number of Hidden Units) 수가 올라감에 따라 학습셋의 예측률(Average Performance on Training Sets)과 테스트셋의 예측률(Average Performance on Testing Sets)이 어떻게 변하는지임
- 이 부분만 따로 뽑아 정리하면 표 13-1과 같음

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

표 13-1 은닉층 수의 변화에 따른 학습셋 및 테스트셋의 예측률

3 | 학습셋과 테스트셋

- 은닉층이 늘어날수록 학습셋의 예측률이 점점 올라가다가 결국 24개 층에 이르면 100%의 예측률을 보임

- 하지만 이 모델을 토대로 테스트한 결과는 어떤가?
- 테스트셋 예측률은 12개에서 90.4%로 최고를 이루다 24개째에선 다시 89.2%로 떨어지고 맒
- 즉, 식이 복잡해지고 학습량이 늘어날수록 학습 데이터를 통한 예측률은 계속해서 올라가지만, 테스트셋을 이용한 예측률은 오히려 떨어짐

3 | 학습셋과 테스트셋

- 주어진 데이터를 학습셋과 테스트셋으로 나누는 예제를 만들어 보자

- 불러온 X 데이터와 Y 데이터에서 각각 정해진 비율(%)만큼 구분하여 한 그룹은 학습에 사용하고 다른 한 그룹은 테스트에 사용하게 하는 함수가 sklearn 라이브러리의 train_test_split() 함수

- 따라서 다음과 같이 학습셋과 테스트셋을 만들 수 있음
- 다음은 학습셋을 70%, 테스트셋을 30%로 설정했을 때의 예

```
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)
```

3 | 학습셋과 테스트셋

- 모델을 실행하는 부분에서 위에서 만들어진 학습셋으로 학습을, 테스트셋으로 테스트를 하게 하려면 다음과 같이 실행

```
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]))
```

3 | 학습셋과 테스트셋

코드 13-2 초음파 광물 예측하기: 학습셋과 테스트셋 구분

- **예제 소스** deep_code/05_Sonar_Train_Test.py

```
from keras.models import Sequential
from keras.layers.core import Dense
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split

import pandas as pd
import numpy
import tensorflow as tf

# seed 값 설정
seed = 0
numpy.random.seed(seed)
tf.set_random_seed(seed)
df = pd.read_csv('../dataset/sonar.csv', header=None)
```

3 | 학습셋과 테스트셋

```
dataset = df.values
X = dataset[:,0:60]
Y_obj = dataset[:,60]

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

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

model = Sequential()
model.add(Dense(24, input_dim=60, activation='relu'))
model.add(Dense(10, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

3 | 학습셋과 테스트셋

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

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]))
```

3 | 학습셋과 테스트셋

- 실행 결과

```
(중략)

Epoch 127/130
145/145 [==============================] - 0s - loss: 0.0284 - acc: 0.9862
Epoch 128/130
145/145 [==============================] - 0s - loss: 0.0275 - acc: 0.9931
Epoch 129/130
145/145 [==============================] - 0s - loss: 0.0261 - acc: 0.9862
Epoch 130/130
145/145 [==============================] - 0s - loss: 0.0237 - acc: 0.9931
32/63 [==============>..............] - ETA: 0s
 Test Accuracy: 0.8095
```

- 테스트셋으로 실험해 본 결과 80.95%의 예측 성공률을 보임
- 앞서(코드 13-1) 학습셋만으로 실험했을 때의 99.52%와 비교해 보면 차이가 남

4 | 모델 저장과 재사용

- 학습이 끝난 후 테스트해 본 결과가 만족스러울 때 이를 모델로 저장하여 새로운 데이터에 사용할 수 있음
- 앞서 학습한 결과를 모델로 저장하려면 다음과 같이 실행

```
from keras.models import load_model

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

- 이를 불러오려면 다음과 같이 실행

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

4 | 모델 저장과 재사용

코드 13-3 초음파 광물 예측하기: 모델 저장과 재사용

- **예제 소스** deep_code/06-Sonar-Save-Model.py

```
from keras.models import Sequential, load_model
from keras.layers.core import Dense
from sklearn.preprocessing import LabelEncoder

import pandas as pd
import numpy
import tensorflow as tf

# seed 값 설정
seed = 0
numpy.random.seed(seed)
tf.set_random_seed(seed)

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

4 | 모델 저장과 재사용

```
dataset = df.values
X = dataset[:,0:60]
Y_obj = dataset[:,60]

e = LabelEncoder()
e.fit(Y_obj)
Y = e.transform(Y_obj)
# 학습셋과 테스트셋을 나눔
from sklearn.model_selection import train_test_split

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.3,
random_state=seed)

model = Sequential()
model.add(Dense(24, input_dim=60, activation='relu'))
model.add(Dense(10, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

4 | 모델 저장과 재사용

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

model.fit(X_train, Y_train, epochs=130, batch_size=5)
model.save('my_model.h5')   # 모델을 컴퓨터에 저장

del model       # 테스트를 위해 메모리 내의 모델을 삭제
model = load_model('my_model.h5') # 모델을 새로 불러옴

print("\n Test Accuracy: %.4f" % (model.evaluate(X_test, Y_test)[1]))  # 불러온 모델로 테스트 실행
```

5 | k겹 교차 검증

- 앞서 가지고 있는 데이터의 약 70%를 학습셋으로 써야 했으므로 테스트셋은 겨우 전체 데이터의 30%에 그쳤음
- 이 정도 테스트만으로는 실제로 얼마나 잘 작동하는지 확신하기는 쉽지 않음

5 | k겹 교차 검증

- 이러한 단점을 보완하고자 만든 방법이 바로 k겹 교차 검증(k-fold cross validation)
- 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 예가 그림 13-5에 설명되어 있음

5 | k겹 교차 검증

그림 13-5 5겹 교차 검증의 도식

5 | k겹 교차 검증

- k겹 교차 검증 만들어 보기

- 데이터를 원하는 숫자만큼 쪼개 각각 학습셋과 테스트셋으로 사용되게 만드는 함수는 sklearn의 StratifiedKFold()

```
from sklearn.model_selection import StratifiedKFold

n_fold = 10
skf = StratifiedKFold(n_splits=n_fold, shuffle=True, random_state=seed)
```

5 | k겹 교차 검증

- 10개의 파일로 쪼개 테스트하는 10-fold cross validation을 실시하도록 n_ fold의 값을 10으로 설정한 뒤 StratifiedKFold() 함수에 적용했음
- 그런 다음 모델을 만들고 실행하는 부분을 for 구문으로 묶어 n_fold만큼 반복되게 함

```
for train, test in skf.split(X, Y):
    model = Sequential()
    model.add(Dense(24, input_dim=60, activation='relu'))
    model.add(Dense(10, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='mean_squared_error',
                  optimizer='adam',
                  metrics=['accuracy'])
    model.fit(X[train], Y[train], epochs=100, batch_size=5)
```

5 | k겹 교차 검증

- 정확도(Accuracy)를 매번 저장하여 한 번에 보여줄 수 있게 accuracy 배열을 만들어 보자

```
accuracy = []

for train, test in skf.split(X, Y):

(생략)

    k_accuracy = "%.4f" % (model.evaluate(X[test], Y[test])[1])
    accuracy.append(k_accuracy)

print("\n %.f fold accuracy:" % n_fold, accuracy)
```

- 이를 종합하여 10-fold cross validation이 포함된 스크립트를 다음과 같이 완성할 수 있음

5 | k겹 교차 검증

코드 13-4 초음파 광물 예측하기: k겹 교차 검증

- **예제 소스** deep_code/07_Sonar-K-fold.py

```
from keras.models import Sequential
from keras.layers.core import Dense
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import StratifiedKFold

import numpy
import pandas as pd
import tensorflow as tf

# seed 값 설정
seed = 0
numpy.random.seed(seed)
tf.set_random_seed(seed)
```

5 | k겹 교차 검증

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

dataset = df.values
X = dataset[:,0:60]
Y_obj = dataset[:,60]

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

# 10개의 파일로 쪼갬
n_fold = 10
skf = StratifiedKFold(n_splits=n_fold, shuffle=True, random_state=seed)

# 빈 accuracy 배열
accuracy = []
```

5 | k겹 교차 검증

```
# 모델의 설정, 컴파일, 실행
for train, test in skf.split(X, Y):
    model = Sequential()
    model.add(Dense(24, input_dim=60, activation='relu'))
    model.add(Dense(10, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    model.compile(loss='mean_squared_error',
                  optimizer='adam',
                  metrics=['accuracy'])
    model.fit(X[train], Y[train], epochs=100, batch_size=5)
    k_accuracy = "%.4f" % (model.evaluate(X[test], Y[test])[1])
    accuracy.append(k_accuracy)

# 결과 출력
print("\n %.f fold accuracy:" % n_fold, accuracy)
```

5 | k겹 교차 검증

- 실행 결과

```
(중략)

Epoch 98/100
188/188 [==============================] - 0s - loss: 0.0319 - acc: 0.9787
Epoch 99/100
188/188 [==============================] - 0s - loss: 0.0298 - acc: 0.9787
Epoch 100/100
188/188 [==============================] - 0s - loss: 0.0336 - acc: 0.9628
20/20 [==============================] - 0s

 10 fold accuracy: ['0.7727', '0.7619', '0.9524', '0.9524', '0.8095', '0.7143',
'0.8571', '0.9500', '0.9000', '0.8500']
```

- 딥러닝이 잘 구동되는 것을 확인할 수 있음
- 실행 후 10번의 테스트 값들이 출력되었음

14장 베스트 모델 만들기

[실습] 와인 종류 예측

1 | 데이터의 확인과 실행

2 | 모델 업데이트하기

3 | 그래프로 확인하기

4 | 학습의 자동 중단

- **실습 데이터** 와인의 종류 예측 : datset/wine.csv

- 이번 실습을 위해 사용되는 데이터는 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

1 | 데이터의 확인과 실행

- 먼저 df_pre라는 공간에 데이터를 불러옴
- 그런 다음 sample() 함수를 사용하여 원본 데이터의 몇 %를 사용할지를 지정

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

- sample() 함수는 원본 데이터에서 정해진 비율만큼 랜덤으로 뽑아오는 함수
- frac = 1이라고 지정하면 원본 데이터의 100%를 불러오라는 의미
- frac = 0.5로 지정하면 50%만 랜덤으로 불러옴
- 원본 데이터를 모두 랜덤으로 불러왔으므로 그중 처음 5줄을 출력해 보자

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

1 | 데이터의 확인과 실행

	0	1	2	3	4	5	6	7	8	9	10	11	12
964	8.5	0.47	0.27	1.9	0.058	18	38	0.99518	3.16	0.85	11.1	6	1
664	12.1	0.4	0.52	2	0.092	15	54	1	3.03	0.66	10.2	5	1
1692	6.9	0.21	0.33	1.8	0.034	48	136	0.9899	3.25	0.41	12.6	7	0
5801	6.7	0.24	0.31	2.3	0.044	37	113	0.99013	3.29	0.46	12.9	6	0
2207	6.1	0.28	0.25	17.75	0.044	48	161	0.9993	3.34	0.48	9.5	5	0

- 한 줄당 모두 13개의 정보가 있음(0~12)
- 이어서 전체 정보를 출력해 보자

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

1 | 데이터의 확인과 실행

Data	columns (total 13 columns):		
0	6497	non-null	float64
1	6497	non-null	float64
2	6497	non-null	float64
3	6497	non-null	float64
4	6497	non-null	float64
5	6497	non-null	float64
6	6497	non-null	float64
7	6497	non-null	float64
8	6497	non-null	float64
9	6497	non-null	float64
10	6497	non-null	float64
11	6497	non-null	int64
12	6497	non-null	int64
dtypes: float64(11), int64(2)			
memory usage: 710.6 KB			

1 | 데이터의 확인과 실행

- 총 6497개의 샘플이 있음을 알 수 있음
- 13개의 속성이 각각 무엇인지는 데이터를 내려받은 UCI 머신러닝 저장소에서 확인할 수 있음
- 이곳에 옮겨보면 다음과 같음

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		

1 | 데이터의 확인과 실행

- 이제 0~11까지에 해당하는 12개의 정보를 가지고 13번째 클래스를 맞추는 과제 임을 확인하였음
- 이 정보를 토대로 X 값과 Y 값을 다음과 같이 정함

```
dataset = df.values
X = dataset[:,0:12]
Y = dataset[:,12]
```

1 | 데이터의 확인과 실행

- 이제 딥러닝을 실행할 차례

- 4개의 은닉층을 만들어 각각 30, 12, 8, 1개의 노드를 줌
- 이항 분류(binary classification) 문제이므로 오차 함수는 binary_crossentropy를 사용하고 최적화 함수로 adam을 사용
- 전체 샘플이 200회 반복되어 입력될 때까지 실험을 반복
- 한 번에 입력되는 입력 값은 200개씩 되게끔 해서 종합하면 다음과 같은 프로그램이 완성됨

1 | 데이터의 확인과 실행

코드 14-1 와인의 종류 예측하기: 데이터 확인과 실행

- **예제 소스** deep_code/08_Wine.py

```
from keras.models import Sequential
from keras.layers import Dense
from keras.callbacks import ModelCheckpoint, EarlyStopping

import pandas as pd
import numpy
import tensorflow as tf
import matplotlib.pyplot as plt

# seed 값 설정
seed = 0
numpy.random.seed(seed)
tf.set_random_seed(seed)
```

1 | 데이터의 확인과 실행

```
# 데이터 입력
df_pre = pd.read_csv('../dataset/wine.csv', header=None)
df = df_pre.sample(frac=1)
dataset = df.values
X = dataset[:,0:12]
Y = dataset[:,12]

# 모델 설정
model = Sequential()
model.add(Dense(30, input_dim=12, activation='relu'))
model.add(Dense(12, activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
```

1 | 데이터의 확인과 실행

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

# 모델 실행
model.fit(X, Y, epochs=200, batch_size=200)

# 결과 출력
print("\n Accuracy: %.4f" % (model.evaluate(X, Y)[1]))
```

1 | 데이터의 확인과 실행

- 실행 결과

```
Epoch 198/200
 200/6497 [..............................] - ETA: 0s - loss: 0.0944 - acc: 0.9700
6497/6497 [==============================] - 0s - loss: 0.0461 - acc: 0.9883
Epoch 199/200
 200/6497 [..............................] - ETA: 0s - loss: 0.0472 - acc: 0.9900
6497/6497 [==============================] - 0s - loss: 0.0477 - acc: 0.9872
Epoch 200/200
 200/6497 [..............................] - ETA: 0s - loss: 0.0214 - acc: 1.0000
6497/6497 [==============================] - 0s - loss: 0.0464 - acc: 0.9886
 32/6497 [..............................] - ETA: 4s
2656/6497 [===========>..................] - ETA: 0s
5184/6497 [======================>.......] - ETA: 0s
 Accuracy: 0.9883
```

- 정확도가 98.83%인 딥러닝 프레임워크를 완성

2 | 모델 업데이트하기

- 에포크(epoch)마다 모델의 정확도를 기록하면서 저장해 보자

- 먼저 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

2 | 모델 업데이트하기

```
import os

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

modelpath="./model/{epoch:02d}-{val_loss:.4f}.hdf5"
```

- 모델을 저장하기 위해 케라스의 콜백 함수 중 ModelCheckpoint 함수를 불러옴

```
from keras.callbacks import ModelCheckpoint
```

2 | 모델 업데이트하기

- 그리고 checkpointer라는 변수를 만들어 이곳에 모니터할 값을 지정
- 테스트 오차는 케라스 내부에서 val_loss로 기록됨(참고로 학습 정확도는 acc, 테스트셋 정확도는 val_acc, 학습셋 오차는 loss로 각각 기록됨)
- 모델이 저장될 곳을 앞서 만든 modelpath로 지정하고 verbose의 값을 1로 정하면 해당 함수의 진행 사항이 출력되고, 0으로 정하면 출력되지 않음

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

2 | 모델 업데이트하기

- 이제 모델을 학습할 때마다 위에서 정한 checkpointer의 값을 받아 지정된 곳에 모델을 저장

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

- 실행하면 다음과 같음

```
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
```

2 | 모델 업데이트하기

- Epoch 0~199까지 총 200개의 모델이 model 폴더에 저장되었음
- 저장된 파일의 이름이 곧 에포크 수와 이때의 테스트셋 오차 값
- 이때 ModelCheckpoint() 함수에 모델이 앞서 저장한 모델보다 나아졌을 때만 저장하게끔 하려면 save_best_only 값을 True로 지정

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

2 | 모델 업데이트하기

코드 14-2 와인의 종류 예측하기: 모델 업데이트

- **예제 소스** deep_code/09_Wine_Checkpoint.py

```
from keras.models import Sequential
from keras.layers import Dense
from keras.callbacks import ModelCheckpoint

import pandas as pd
import numpy
import os
import tensorflow as tf

# seed 값 설정
seed = 0
numpy.random.seed(seed)
tf.set_random_seed(seed)
```

2 | 모델 업데이트하기

```
df_pre = pd.read_csv('../dataset/wine.csv', header=None)
df = df_pre.sample(frac=1)
dataset = df.values
X = dataset[:,0:12]
Y = dataset[:,12]

# 모델 설정
model = Sequential()
model.add(Dense(30, input_dim=12, activation='relu'))
model.add(Dense(12, activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

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

2 | 모델 업데이트하기

```
# 모델 저장 폴더 설정
MODEL_DIR = './model/'
if not os.path.exists(MODEL_DIR):
    os.mkdir(MODEL_DIR)

# 모델 저장 조건 설정
modelpath="./model/{epoch:02d}-{val_loss:.4f}.hdf5"
checkpointer = ModelCheckpoint(filepath=modelpath, monitor='val_loss', verbose=1,
save_best_only=True)

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

2 | 모델 업데이트하기

- 실행 결과

```
(중략)

Epoch 00183: val_loss improved from 0.04916 to 0.04863, saving model to ./
model/183-0.0486.hdf5
Epoch 00184: val_loss did not improve
Epoch 00185: val_loss did not improve
Epoch 00186: val_loss did not improve
Epoch 00187: val_loss did not improve
Epoch 00188: val_loss did not improve
Epoch 00189: val_loss did not improve
Epoch 00190: val_loss did not improve
Epoch 00191: val_loss did not improve
Epoch 00192: val_loss did not improve
Epoch 00193: val_loss improved from 0.04863 to 0.04855, saving model to ./
model/193-0.0486.hdf5
```

2 | 모델 업데이트하기

```
Epoch 00194: val_loss did not improve
Epoch 00195: val_loss did not improve
Epoch 00196: val_loss did not improve
Epoch 00197: val_loss did not improve
Epoch 00198: val_loss improved from 0.04855 to 0.04823, saving model to ./
model/198-0.0482.hdf5
Epoch 00199: val_loss did not improve
```

- 에포크가 진행되면서 모든 값이 저장되는 것이 아니라 테스트 오차를 실행한 결괏값이 향상되었을 때만 저장되는 것을 볼 수 있음

3 | 그래프로 확인하기

- 에포크를 얼마나 지정할지를 결정하는 것이 중요하다

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

- 에포크에 따른 정확도와 테스트 결과를 그래프를 통해 확인해 보자
- 다음과 같이 스크립트를 작성

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

history = model.fit(X, Y, validation_split=33, epochs=3500, batch_size=500)
```

3 | 그래프로 확인하기

- 모델이 학습되는 과정을 history 변수를 만들어 저장
- 긴 학습의 예를 지켜 보기 위해 에포크(epoch)를 3500으로 조정하였음
- 시간이 너무 오래 걸리지 않도록 sample() 함수를 이용하여 전체 샘플 중 15%만 불러오게 하고, 배치 크기는 500으로 늘려 한 번 딥러닝을 가동할 때 더 많이 입력되게끔 함
- 불러온 샘플 중 33%는 분리하여 테스트셋으로 사용함

3 | 그래프로 확인하기

- 그래프로 표현하기 위한 라이브러리를 불러오고 오차와 정확도의 값을 정함
- y_vloss에 테스트셋(33%)으로 실험한 결과의 오차값을 저장
- y_acc에 학습셋(67%)으로 측정한 정확도의 값을 저장

```
import matplotlib.pyplot as plt

y_vloss=history.history['val_loss']
y_acc=history.history['acc']
```

- x 값을 지정하고 정확도를 파란색으로, 오차를 빨간색으로 표시해 보자

```
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)
```

3 | 그래프로 확인하기

코드 14-3 와인의 종류 예측하기: 그래프 표현

- **예제 소스** deep_code/10_Wine_Overfit_Graph.py

```
from keras.models import Sequential
from keras.layers import Dense
from keras.callbacks import ModelCheckpoint

import pandas as pd
import numpy
import os
import tensorflow as tf
import matplotlib.pyplot as plt

# seed 값 설정
seed = 0
numpy.random.seed(seed)
tf.set_random_seed(seed)
```

3 | 그래프로 확인하기

```
df_pre = pd.read_csv('../dataset/wine.csv', header=None)
df = df_pre.sample(frac=0.15)
dataset = df.values
X = dataset[:,0:12]
Y = dataset[:,12]

# 모델 설정
model = Sequential()
model.add(Dense(30, input_dim=12, activation='relu'))
model.add(Dense(12, activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

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

3 | 그래프로 확인하기

```
# 모델 저장 폴더 설정
MODEL_DIR = './model/'
if not os.path.exists(MODEL_DIR):
    os.mkdir(MODEL_DIR)

# 모델 저장 조건 설정
modelpath="./model/{epoch:02d}-{val_loss:.4f}.hdf5"
checkpointer = ModelCheckpoint(filepath=modelpath, monitor='val_loss', verbose=1,
save_best_only=True)

# 모델 실행 및 저장
history = model.fit(X, Y, validation_split=33, epochs=3500, batch_size=500)

# y_vloss에 테스트셋으로 실험 결과의 오차 값을 저장
y_vloss=history.history['val_loss']
```

3 | 그래프로 확인하기

```
# y_acc에 학습셋으로 측정한 정확도의 값을 저장
y_acc=history.history['acc']

# x 값을 지정하고 정확도를 파란색으로, 오차를 빨간색으로 표시
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)

plt.show()
```

3 | 그래프로 확인하기

- 실행 결과

```
Epoch 3497/3500
653/653 [==============================] - 0s - loss: 0.0119 - acc:
0.9954 - val_loss: 0.0976 - val_acc: 0.9783
Epoch 3498/3500
653/653 [==============================] - 0s - loss: 0.0102 - acc:
0.9985 - val_loss: 0.0885 - val_acc: 0.9783
Epoch 3499/3500
653/653 [==============================] - 0s - loss: 0.0123 - acc:
0.9954 - val_loss: 0.0933 - val_acc: 0.9783
Epoch 3500/3500
653/653 [==============================] - 0s - loss: 0.0097 - acc:
0.9969 - val_loss: 0.0958 - val_acc: 0.9783
```

3 | 그래프로 확인하기

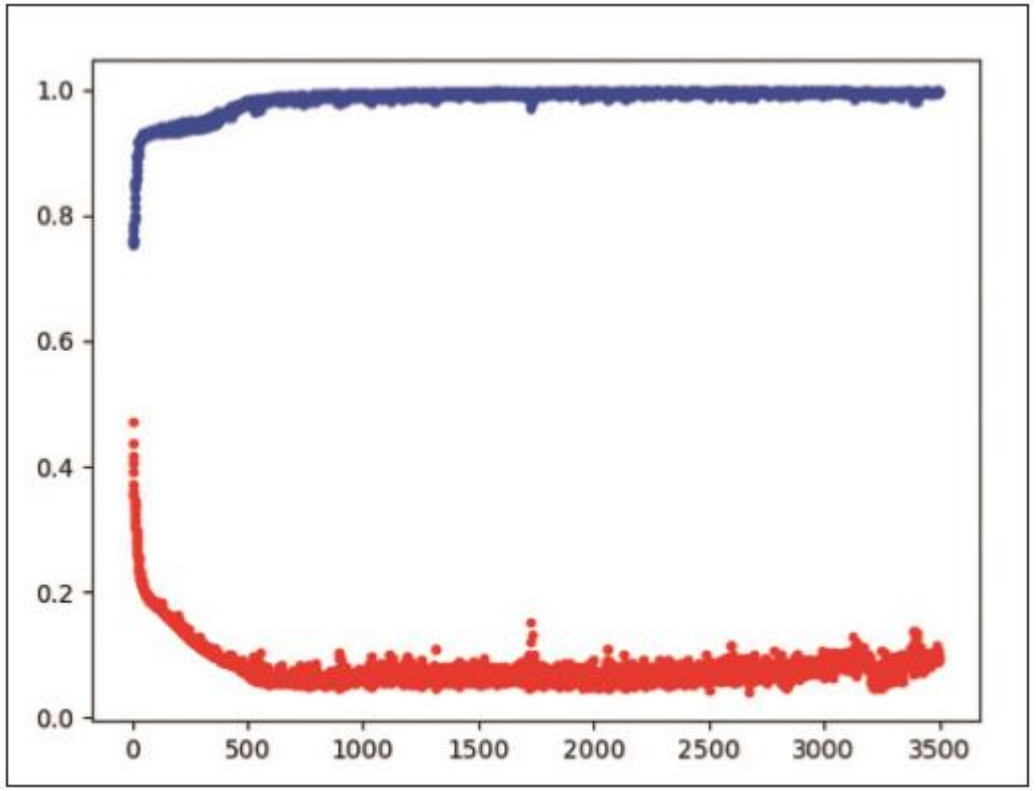

그림 14-1 학습 진행에 따른 학습셋과 테스트셋의 정확도 그래프

3 | 그래프로 확인하기

- 실행 결과로 얻은 그래프를 좀 더 보기 좋게 단순화해 보면 다음과 같다.

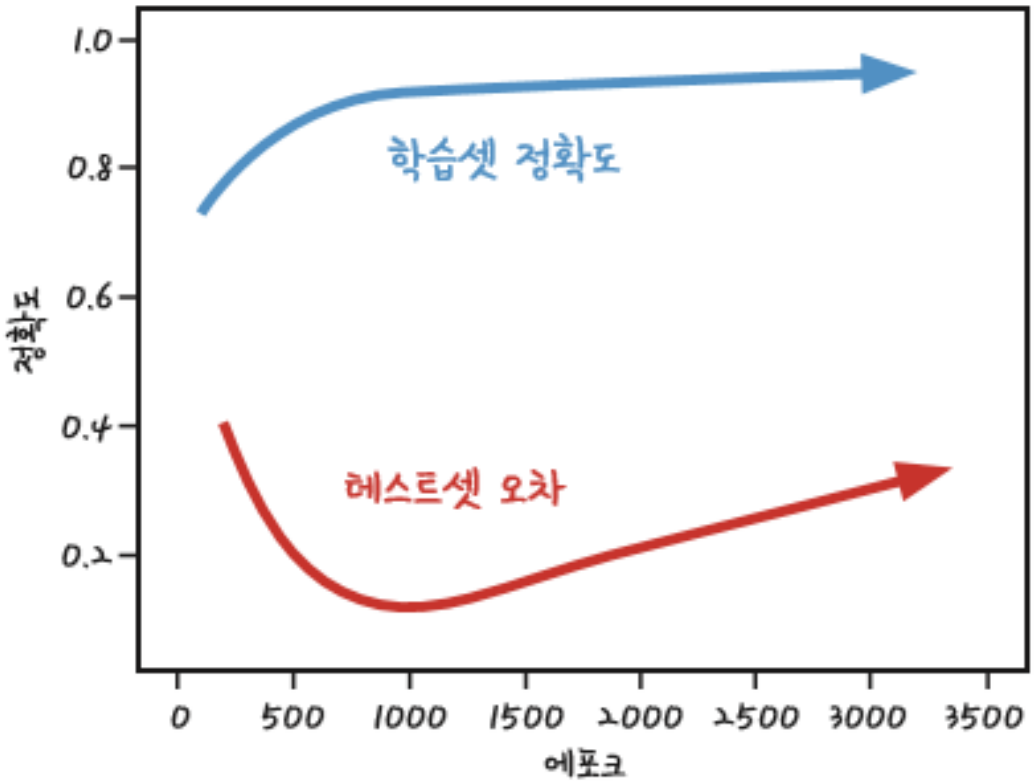

그림 14-2 학습이 진행될수록 학습셋의 정확도는 오르지만 테스트셋에서는 과적합 발생

3 | 그래프로 확인하기

- 앞서 공부한 대로 학습셋의 정확도는 시간이 흐를수록 좋아짐
- 하지만 테스트 결과는 어느 정도 이상 시간이 흐르면 더 나아지지 않는 것을 그래프로 확인할 수 있다!

4 | 학습의 자동 중단

- 학습이 진행될수록 학습셋의 정확도는 올라가지만 과적합으로 인해 테스트셋의 실험 결과는 점점 나빠지게 됨
- 케라스의 EarlyStopping() 함수는 이렇게 학습이 진행되어도 테스트셋 오차가 줄지 않으면 **학습을 멈추게 하는** 함수
- 이를 적용하기 위해 케라스에서 해당 함수를 불러오자

```
from keras.callbacks import EarlyStopping
```

4 | 학습의 자동 중단

- arlyStopping() 함수에 모니터할 값과 테스트 오차가 좋아지지 않아도 몇 번까지 기다릴지를 정함
- 이를 early_stopping_callback에 저장함

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

- 이제 앞서 정한 그대로 에포크 횟수와 배치 크기 등을 설정하고 early_stopping_callback 값을 불러옴

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

4 | 학습의 자동 중단

코드 14-4 와인의 종류 예측하기: 학습의 자동 중단

- **예제 소스** deep_code/12_Wine_Check_and_Stop.py

```
from keras.models import Sequential
from keras.layers import Dense
from keras.callbacks import EarlyStopping

import pandas as pd
import numpy
import tensorflow as tf

# seed 값 설정
seed = 0
numpy.random.seed(seed)
tf.set_random_seed(seed)
```

4 | 학습의 자동 중단

```
df_pre = pd.read_csv('../dataset/wine.csv', header=None)
df = df_pre.sample(frac=0.15)

dataset = df.values
X = dataset[:,0:12]
Y = dataset[:,12]

model = Sequential()
model.add(Dense(30, input_dim=12, activation='relu'))
model.add(Dense(12, activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])
```

4 | 학습의 자동 중단

```
# 자동 중단 설정
early_stopping_callback = EarlyStopping(monitor='val_loss', patience=100)

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

# 결과 출력
print("\n Accuracy: %.4f" % (model.evaluate(X, Y)[1]))
```

4 | 학습의 자동 중단

- 실행 결과

```
(중략)

Epoch 775/2000
780/780 [==============================] - 0s - loss: 0.0186 - acc:
0.9962 - val_loss: 0.0662 - val_acc: 0.9795
Epoch 776/2000
780/780 [==============================] - 0s - loss: 0.0193 - acc:
0.9936 - val_loss: 0.0743 - val_acc: 0.9795
Epoch 777/2000
780/780 [==============================] - 0s - loss: 0.0198 - acc:
0.9936 - val_loss: 0.0660 - val_acc: 0.9795
Epoch 778/2000
780/780 [==============================] - 0s - loss: 0.0236 - acc:
0.9936 - val_loss: 0.0699 - val_acc: 0.9846
 32/975 [..............................] - ETA: 0s
 Accuracy: 0.9918
```

4 | 학습의 자동 중단

- 에포크를 2000으로 설정하였지만, 도중에 계산이 멈추는 것을 확인할 수 있음
- 두 함수, 즉 모델 업데이트 함수와 학습 자동 중단 함수를 동시에 사용해 보자
- 코드 14-5와 같음

4 | 학습의 자동 중단

코드 14-5 와인의 종류 예측하기: 전체 코드

- **예제 소스** deep_code/12_Wine_Check_and_Stop.py

```
from keras.models import Sequential
from keras.layers import Dense
from keras.callbacks import ModelCheckpoint, EarlyStopping

import pandas as pd
import numpy
import os
import tensorflow as tf

# seed 값 설정
seed = 0
numpy.random.seed(seed)
tf.set_random_seed(seed)
```

4 | 학습의 자동 중단

```
df_pre = pd.read_csv('../dataset/wine.csv', header=None)
df = df_pre.sample(frac=0.15)
dataset = df.values
X = dataset[:,0:12]
Y = dataset[:,12]

model = Sequential()
model.add(Dense(30, input_dim=12, activation='relu'))
model.add(Dense(12, activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

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

4 | 학습의 자동 중단

```
# 모델 저장 폴더 만들기
MODEL_DIR = './model/'
if not os.path.exists(MODEL_DIR):
    os.mkdir(MODEL_DIR)

modelpath="./model/{epoch:02d}-{val_loss:.4f}.hdf5"

# 모델 업데이트 및 저장
checkpointer = ModelCheckpoint(filepath=modelpath, monitor='val_loss', verbose=1,
save_best_only=True)

# 학습 자동 중단 설정
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])
```

4 | 학습의 자동 중단

- 실행 결과

```
(중략)

Epoch 00680: val_loss did not improve
Epoch 00681: val_loss did not improve
Epoch 00682: val_loss improved from 0.05415 to 0.05286, saving model to ./
model/682-0.0529.hdf5
Epoch 00683: val_loss did not improve
Epoch 00684: val_loss did not improve
Epoch 00685: val_loss did not improve

(중략)

Epoch 00781: val_loss did not improve
Epoch 00782: val_loss did not improve
Epoch 00783: val_loss did not improve
```

4 | 학습의 자동 중단

- 전체 샘플의 15%만 사용했으므로 780개의 샘플이 입력되었음
- 전체 반복 값을 3500으로 하였으나 마지막 업데이트는 이번 실험의 경우 682회째였음
- 이후 100번가량 모델이 나아지지 않자 학습이 자동 중단되었음
- 이때의 모델은 model 폴더에 682-0.0529.hdf5라는 이름으로 저장됨

15장 선형 회귀 적용하기

[실습] 보스턴 집값 예측

1 | 데이터 확인하기

2 | 선형 회귀 실행

- **실습 데이터** 보스턴 집값 예측 : dataset/housing.csv

- 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

- 지금까지 실습의 예로 삼았던 데이터들은 참(1) 또는 거짓(0)을 맞히는 문제, 아니면 여러 개의 보기 중 맞는 하나를 예측하는 문제였음

- 그런데 이번에는 하나의 정답을 맞히는 것이 아니라 수치를 예측하는 문제
  - → 주어진 환경 요인과 집값의 변동을 학습해서 환경 요인만 놓고 집값을 예측하는 것
  - → 선형 회귀 문제!

1 | 데이터 확인하기

- 먼저 데이터를 확인해 보자

```
import pandas as pd

df = pd.read_csv("../dataset/housing.csv", delim_whitespace=True, header=None)

print(df.info())
```

1 | 데이터 확인하기

Range Index:506 entries,0 to 505			
Data columns (total 14 columns):			
0	506	non-null	float64
1	506	non-null	float64
2	506	non-null	float64
3	506	non-null	int64
...	...	...	...
11	506	non-null	float64
12	506	non-null	float64
13	506	non-null	float64
Dtypes: float64(12), int64(2)			
memory usage: 55.4 KB			

1 | 데이터 확인하기

- Index가 506개이므로 총 샘플의 수는 506개이고, 컬럼 수는 14개이므로 13개의 속성과 1개의 클래스로 이루어졌음을 짐작할 수 있음
- 일부를 출력해서 확인해 보자

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

	0	1	2	3	···	12	13
0	0.00632	18.0	2.31	0	···	4.98	24.0
1	0.02731	0	7.07	0	···	9.14	21.6
2	0.02729	0	7.07	0	···	4.03	34.7
3	0.03237	0	2.18	0	···	2.94	33.4
4	0.06905	0	2.18	0	···	5.33	36.2

1 | 데이터 확인하기

- 마지막 컬럼을 보면 지금까지와는 다름
- 클래스로 구분된 게 아니라 가격이 나와 있음
- 각 속성이 가리키는 바는 다음과 같음

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)

2 | 선형 회귀 실행

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

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

2 | 선형 회귀 실행

- 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))
```

- flatten()은 데이터 배열이 몇 차원이든 모두 1차원으로 바꿔 읽기 쉽게 해 주는 함수
- range('숫자')는 0부터 '숫자-1'만큼 차례대로 증가하며 반복 되는 값을 만듦
- 즉, range(10)은 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]를 말함
- for in 구문을 사용해서 10번 반복하게 하였음

2 | 선형 회귀 실행

코드 15-1 보스턴 집값 예측하기

- **예제 소스** deep_code/13_Boston.py

```
from keras.models import Sequential
from keras.layers import Dense
from sklearn.model_selection import train_test_split

import numpy
import pandas as pd
import tensorflow as tf

# seed 값 설정
seed = 0
numpy.random.seed(seed)
tf.set_random_seed(seed)

df = pd.read_csv("../dataset/housing.csv", delim_whitespace=True, header=None)
```

2 | 선형 회귀 실행

```
dataset = df.values
X = dataset[:,0:13]
Y = dataset[:,13]
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.3,
random_state=seed)

model = Sequential()
model.add(Dense(30, input_dim=13, activation='relu'))
model.add(Dense(6, activation='relu'))
model.add(Dense(1))

model.compile(loss='mean_squared_error',
              optimizer='adam')

model.fit(X_train, Y_train, epochs=200, batch_size=10)
```

2 | 선형 회귀 실행

```
# 예측 값과 실제 값의 비교
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))
```

2 | 선형 회귀 실행

- 실행 결과

```
실제가격: 22.600, 예상가격: 20.701
실제가격: 50.000, 예상가격: 26.329
실제가격: 23.000, 예상가격: 21.918
실제가격: 8.300, 예상가격: 12.454
실제가격: 21.200, 예상가격: 18.575
실제가격: 19.900, 예상가격: 21.978
실제가격: 20.600, 예상가격: 19.141
실제가격: 18.700, 예상가격: 24.095
실제가격: 16.100, 예상가격: 19.012
실제가격: 18.600, 예상가격: 13.672
```

- 실제가격과 예상가격이 비례하여 변화하는 것을 확인할 수 있음