넷째마당
딥러닝 기본기 다지기

14장 모델 성능 향상시키기

1 데이터의 확인과 검증셋

2 모델 업데이트하기

3 그래프로 과적합 확인하기

4 학습의 자동 중단

1 데이터의 확인과 검증셋

1 데이터의 확인과 검증셋

- **데이터의 확인과 검증셋**
  - 먼저 데이터를 불러와 대략적인 구조를 살펴보자

```
import pandas as pd

# 깃허브에 준비된 데이터를 가져옵니다.
!git clone https://github.com/taehojo/data.git

# 와인 데이터를 불러옵니다.
df = pd.read_csv('./data/wine.csv', header=None)

# 데이터를 미리 보겠습니다.
df
```

1 데이터의 확인과 검증셋

- 데이터의 확인과 검증셋

실행 결과

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.99780	3.51	0.56	9.4	5	1
1	7.8	0.88	0.00	2.6	0.098	25.0	67.0	0.99680	3.20	0.68	9.8	5	1
2	7.8	0.76	0.04	2.3	0.092	15.0	54.0	0.99700	3.26	0.65	9.8	5	1
3	11.2	0.28	0.56	1.9	0.075	17.0	60.0	0.99800	3.16	0.58	9.8	6	1
4	7.4	0.70	0.00	1.9	0.076	11.0	34.0	0.99780	3.51	0.56	9.4	5	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...
6492	6.2	0.21	0.29	1.6	0.039	24.0	92.0	0.99114	3.27	0.50	11.2	6	0
6493	6.6	0.32	0.36	8.0	0.047	57.0	168.0	0.99490	3.15	0.46	9.6	5	0
6494	6.5	0.24	0.19	1.2	0.041	30.0	111.0	0.99254	2.99	0.46	9.4	6	0
6495	5.5	0.29	0.30	1.1	0.022	20.0	110.0	0.98869	3.34	0.38	12.8	7	0
6496	6.0	0.21	0.38	0.8	0.020	22.0	98.0	0.98941	3.26	0.32	11.8	6	0

6497 rows × 13 columns

1 데이터의 확인과 검증셋

- **데이터의 확인과 검증셋**
  - 샘플이 전체 6,497개 있음
  - 모두 속성이 12개 기록되어 있고 13번째 열에 클래스가 준비되어 있음
  - 각 속성에 대한 정보는 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	클래스(1: 레드 와인, 0: 화이트 와인)
6	총 아황산 농도		

1 데이터의 확인과 검증셋

- **데이터의 확인과 검증셋**
  - 0~11번째 열에 해당하는 속성 12개를 X로, 13번째 열을 y로 정하겠음

```
X = df.iloc[:,0:12]
y = df.iloc[:,12]
```

1 데이터의 확인과 검증셋

- **데이터의 확인과 검증셋**
  - 이제 딥러닝을 실행할 차례
  - 앞서 우리는 학습셋과 테스트셋을 나누는 방법에 대해 알아보았음
  - 이 장에서는 여기에 검증셋을 더해 보자

1 데이터의 확인과 검증셋

▼ 그림 14-1 | 학습셋, 테스트셋, 검증셋

1 데이터의 확인과 검증셋

- **데이터의 확인과 검증셋**
  - 학습이 끝난 모델을 테스트해 보는 것이 테스트셋의 목적이라면, 최적의 학습 파라미터를 찾기 위해 학습 과정에서 사용하는 것이 검증셋
  - 검증셋을 설정하면 검증셋에 테스트한 결과를 추적하면서 최적의 모델을 만들 수 있음
  - 검증셋은 `model.fit()` 함수 안에 `validation_split`이라는 옵션을 주면 만들어짐
  - 그림 14-1과 같이 전체의 80%를 학습셋으로 만들고 이 중 25%를 검증셋으로 하면 학습셋 : 검증셋 : 테스트셋의 비율이 60 : 20 : 20이 됨

1 데이터의 확인과 검증셋

- **데이터의 확인과 검증셋**
  - 전체 코드를 실행하면 다음과 같음

실습 | 와인의 종류 예측하기: 데이터 확인과 실행

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

# 깃허브에 준비된 데이터를 가져옵니다.
!git clone https://github.com/taehojo/data.git

# 와인 데이터를 불러옵니다.
df = pd.read_csv('./data/wine.csv', header=None)
```

1 데이터의 확인과 검증셋

- 데이터의 확인과 검증셋

```
# 와인의 속성을 X로, 와인의 분류를 y로 저장합니다.
X = df.iloc[:,0:12]
y = df.iloc[:,12]

# 학습셋과 테스트셋으로 나눕니다.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
shuffle=True)

# 모델 구조를 설정합니다.
model = Sequential()
```

1 데이터의 확인과 검증셋

- 데이터의 확인과 검증셋

```
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.summary()

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

# 모델을 실행합니다.
history = model.fit(X_train, y_train, epochs=50, batch_size=500,
validation_split=0.25) # 0.8 x 0.25 = 0.2
```

1 데이터의 확인과 검증셋

- 데이터의 확인과 검증셋

```
# 테스트 결과를 출력합니다.
score = model.evaluate(X_test, y_test)
print('Test accuracy:', score[1])
```

1 데이터의 확인과 검증셋

- 데이터의 확인과 검증셋

실행 결과

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 30)                390
_________________________________________________________________
dense_1 (Dense)              (None, 12)                372
_________________________________________________________________
dense_2 (Dense)              (None, 8)                 104
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 9
=================================================================
Total params: 875
```

1 데이터의 확인과 검증셋

- 데이터의 확인과 검증셋

```
Trainable params: 875
Non-trainable params: 0
_________________________________________________________________

Epoch 1/50
8/8 [==============================] - 1s 23ms/step - loss: 2.9423 - accu
racy: 0.7519 - val_loss: 2.2360 - val_accuracy: 0.7562
... (중략) ...
Epoch 50/50
8/8 [==============================] - 0s 6ms/step - loss: 0.1161 - accura
cy: 0.9574 - val_loss: 0.1523 - val_accuracy: 0.9500
```

1 데이터의 확인과 검증셋

- 데이터의 확인과 검증셋

```
41/41 [==============================] - 0s 1ms/step - loss: 0.1438 - accu
racy: 0.9415
Test accuracy: 0.9415384531021118
```

1 데이터의 확인과 검증셋

- **데이터의 확인과 검증셋**
  - 먼저 세 개의 은닉층을 만들고 각각 30개, 12개, 8개의 노드를 만들었음
  - 50번을 반복했을 때 정확도가 94.15%로 나왔음
  - 꽤 높은 정확도
  - 이것이 과연 최적의 결과일까?
  - 이제 여기에 여러 옵션을 더해 가면서 더 나은 모델을 만들어 가는 방법을 알아보자

2 모델 업데이트하기

2 모델 업데이트하기

- **모델 업데이트하기**
  - 에포크(epochs)는 학습을 몇 번 반복할 것인지 정해 줌
  - 에포크가 50이면 순전파와 역전파를 50번 실시한다는 뜻
  - 학습을 많이 반복한다고 해서 모델 성능이 지속적으로 좋아지는 것은 아님
  - 이를 적절히 정해 주는 것이 중요
  - 만일 50번의 에포크 중 최적의 학습이 40번째에 이루어졌다면, 어떻게 해서 40번째 모델을 불러와 사용할 수 있을까?
  - 이번에는 에포크마다 모델의 정확도를 함께 기록하면서 저장하는 방법을 알아보자

2 모델 업데이트하기

- **모델 업데이트하기**
  - 먼저 모델이 어떤 식으로 저장될지 정함
  - 다음 코드는 ./data/model/all/ 폴더에 모델을 지정
  - 50번째 에포크의 검증셋 정확도가 0.9346이라면 50-0.9346.hdf5라는 이름으로 저장

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

2 모델 업데이트하기

- **모델 업데이트하기**
  - 학습 중인 모델을 저장하는 함수는 케라스 API의 ModelCheckpoint()
  - 모델이 저장될 곳을 정하고 진행되는 현황을 모니터할 수 있도록 verbose는 1(True)로 설정

```
from tensorflow.keras.callbacks import ModelCheckpoint

checkpointer = ModelCheckpoint(filepath=modelpath, verbose=1)
```

2 모델 업데이트하기

- **모델 업데이트하기**
  - 전체 코드를 실행해 보자
  - 앞 절에서 배운 코드 중 model.compile()까지는 동일함
  - 그 아래에 추가되는 코드는 다음과 같음

```
# 모델이 저장되는 조건을 설정합니다.
modelpath = "./data/model/{epoch:02d}-{val_accuracy:.4f}.hdf5"
checkpointer = ModelCheckpoint(filepath=modelpath, verbose=1)

# 모델을 실행합니다.
history = model.fit(X_train, y_train, epochs=50, batch_size=500,
validation_split=0.25, verbose=0, callbacks=[checkpointer])

# 테스트 결과를 출력합니다.
score = model.evaluate(X_test, y_test)
print('Test accuracy:', score[1])
```

2 모델 업데이트하기

- 모델 업데이트하기

실행 결과

```
Epoch 00001: saving model to ./data/model/all\01-0.7646.hdf5
Epoch 00002: saving model to ./data/model/all\02-0.7646.hdf5
... (중략) ...
Epoch 00049: saving model to ./data/model/all\49-0.9408.hdf5
Epoch 00050: saving model to ./data/model/all\50-0.9408.hdf5

41/41 [==============================] - 0s 2ms/step - loss: 0.1686 - accu
racy: 0.9392
Test accuracy: 0.939230740070343
```

2 모델 업데이트하기

- **모델 업데이트하기**
  - 파일명을 통해 에포크 수와 정확도를 알 수 있음
  - 첫 번째 에포크에서 76.46%였던 정확도가 50번째에서 94.08%로 업데이트되는 것과 각 에포크별 모델이 지정된 폴더에 저장되는 것을 볼 수 있음
  - 테스트하면 93.9%의 정확도를 보여 줌
  - 실행 결과는 환경에 따라 미세하게 달라질 수 있음

3 그래프로 과적합 확인하기

3 그래프로 과적합 확인하기

- **그래프로 과적합 확인하기**
  - 역전파를 50번 반복하면서 학습을 진행
  - 과연 이 반복 횟수는 적절했을까?
  - 학습의 반복 횟수가 너무 적으면 데이터셋의 패턴을 충분히 파악하지 못함
  - 학습을 너무 많이 반복하는 것도 좋지 않음
  - 너무 과한 학습은 13.2절에서 이야기한 바 있는 과적합 현상을 불러오기 때문임
  - 적절한 학습 횟수를 정하기 위해서는 검증셋과 테스트셋의 결과를 그래프로 보는 것이 가장 좋음
  - 이를 확인하기 위해 학습을 길게 실행해 보고 결과를 알아보자

3 그래프로 과적합 확인하기

- **그래프로 과적합 확인하기**
  - 먼저 에포크 수를 2000으로 늘려 긴 학습을 해 보자

```
history = model.fit(X_train, y_train, epochs=2000, batch_size=500,
validation_split=0.25)
```

3 그래프로 과적합 확인하기

- **그래프로 과적합 확인하기**
  - 이 코드를 포함해 그동안 model.fit()을 실행할 때마다 결과를 항상 history에 저장해 왔음
  - 이제 저장된 history를 어떻게 활용할 수 있는지 알아보자
  - model.fit()은 학습을 진행하면서 매 에포크마다 결과를 출력
  - 일반적으로 loss 값이 출력되고 model.compile()에서 metrics를 accuracy로 지정하면 accuracy 값이 함께 출력
  - loss는 학습을 통해 구한 예측 값과 실제 값의 차이(=오차)를 의미하고 accuracy는 전체 샘플 중에서 정답을 맞춘 샘플이 몇 개인지의 비율(=정확도)을 의미
  - 이번 예제처럼 검증셋을 지정하면 val_loss가 함께 출력
  - 이때 metrics를 accuracy로 지정하면 accuracy와 함께 val_accuracy 값도 출력
  - val_loss는 학습한 모델을 검증셋에 적용해 얻은 오차이고, val_accuracy는 검증셋으로 얻은 정확도

3 그래프로 과적합 확인하기

- **그래프로 과적합 확인하기**
  - 이 값이 저장된 history는 model.fit()의 결과를 가진 파이썬 객체로, history.params에는 model.fit()의 설정 값들이, history.epoch에는 에포크 정보가 들어 있게 됨
  - 우리에게 필요한 loss, accuracy, val_loss, val_accuracy는 history.history에 들어 있음
  - 이를 판다스 라이브러리로 불러와 내부를 살펴보자

```
hist_df = pd.DataFrame(history.history)
hist_df
```

3 그래프로 과적합 확인하기

- 그래프로 과적합 확인하기

실행 결과

	loss	accuracy	val_loss	val_accuracy
0	0.157924	0.944316	0.173545	0.931538
1	0.156247	0.943546	0.168429	0.933846
2	0.152906	0.942777	0.166696	0.933846
3	0.151120	0.945086	0.165191	0.932308
4	0.148956	0.945856	0.159559	0.936154

3 그래프로 과적합 확인하기

- 그래프로 과적합 확인하기

...	...	...	...	...
1995	0.018356	0.994355	0.066240	0.985385
1996	0.017976	0.994355	0.064675	0.985385
1997	0.018248	0.994098	0.064908	0.985385
1998	0.018649	0.994611	0.065713	0.984615
1999	0.019730	0.993841	0.068250	0.984615

2000 rows × 4 columns

- 2,000번의 학습 결과가 저장되어 있음을 알 수 있음

3 그래프로 과적합 확인하기

- **그래프로 과적합 확인하기**
  - 이 중 학습한 모델을 검증셋에 적용해 얻은 오차(val_loss)는 y_vloss에 저장하고 학습셋에서 얻은 오차(loss)는 y_loss에 저장해 보자

```
y_vloss = hist_df['val_loss']
y_loss = hist_df['loss']
```

3 그래프로 과적합 확인하기

- **그래프로 과적합 확인하기**
  - 이제 그래프로 표시해 보자
  - 학습셋에서 얻은 오차는 빨간색으로, 검증셋에서 얻은 오차는 파란색으로 표시

```
x_len = np.arange(len(y_loss))
plt.plot(x_len, y_vloss, "o", c="red", markersize=2, label='Testset_loss')
plt.plot(x_len, y_loss, "o", c="blue", markersize=2, label='Trainset_loss')

plt.legend(loc='upper right')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.show()
```

3 그래프로 과적합 확인하기

- **그래프로 과적합 확인하기**
  - 이를 하나의 코드로 정리해서 앞서 실행했던 주피터 노트북에 이어 실행해 보자

실습 | 와인의 종류 예측하기: 그래프 표현

```
# 그래프 확인을 위한 긴 학습(컴퓨터 환경에 따라 시간이 다소 걸릴 수 있습니다)
history = model.fit(X_train, y_train, epochs=2000, batch_size=500,
validation_split=0.25)

# history에 저장된 학습 결과를 확인해 보겠습니다.
hist_df = pd.DataFrame(history.history)
hist_df

# y_vloss에 테스트셋의 오차를 저장합니다.
y_vloss = hist_df['val_loss']
```

3 그래프로 과적합 확인하기

- 그래프로 과적합 확인하기

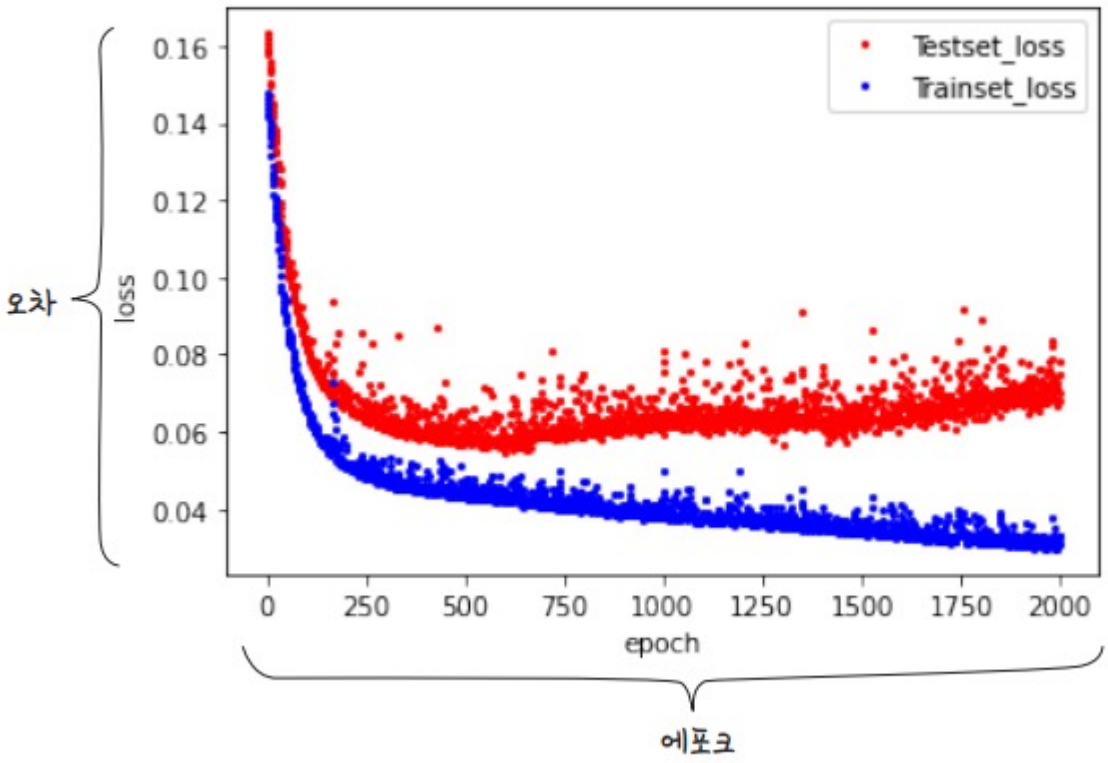
```
# y_loss에 학습셋의 오차를 저장합니다.
y_loss = hist_df['loss']

# x 값을 지정하고 테스트셋의 오차를 빨간색으로, 학습셋의 오차를 파란색으로 표시합니다.
x_len = np.arange(len(y_loss))
plt.plot(x_len, y_vloss, "o", c="red", markersize=2, label='Testset_loss')
plt.plot(x_len, y_loss, "o", c="blue", markersize=2, label='Trainset_loss')

plt.legend(loc='upper right')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.show()
```

3 그래프로 과적합 확인하기

- 그래프로 과적합 확인하기

실행 결과

▼ 그림 14-2 | 학습셋에서 얻은 오차와 검증셋에서 얻은 오차 비교



3 그래프로 과적합 확인하기

- **그래프로 과적합 확인하기**
  - 그래프의 형태는 실행에 따라 조금씩 다를 수 있지만 대략 그림 14-2와 같은 그래프가 나옴
  - 우리가 눈여겨보아야 할 부분은 학습이 오래 진행될수록 검증셋의 오차(파란색)는 줄어들지만 테스트셋의 오차(빨간색)는 다시 커진다는 것
  - 이는 과도한 학습으로 과적합이 발생했기 때문임
  - 이러한 사실을 통해 알 수 있는 것은 검증셋 오차가 커지기 직전까지 학습한 모델이 최적의 횟수로 학습한 모델이라는 것
  - 이제 검증셋의 오차가 커지기 전에 학습을 자동으로 중단시키고, 그때의 모델을 저장하는 방법을 알아보자

4 학습의 자동 중단

4 학습의 자동 중단

- **학습의 자동 중단**
  - 텐서플로에 포함된 케라스 API는 EarlyStopping() 함수를 제공
  - 학습이 진행되어도 테스트셋 오차가 줄어들지 않으면 학습을 자동으로 멈추게 하는 함수
  - 이를 조금 전 배운 ModelCheckpoint() 함수와 함께 사용해 보면서 최적의 모델을 저장해 보자
  - 먼저 다음과 같이 EarlyStopping() 함수를 불러옴

```
from tensorflow.keras.callbacks import EarlyStopping

early_stopping_callback = EarlyStopping(monitor='val_loss', patience=20)
```

4 학습의 자동 중단

- **학습의 자동 중단**
  - monitor 옵션은 model.fit()의 실행 결과 중 어떤 것을 이용할지 정함
  - 검증셋의 오차(val_loss)로 지정
  - patience 옵션은 지정된 값이 몇 번 이상 향상되지 않으면 학습을 종료시킬지 정함
  - monitor='val_loss', patience=20이라고 지정하면 검증셋의 오차가 20번 이상 낮아지지 않을 경우 학습을 종료하라는 의미

4 학습의 자동 중단

- **학습의 자동 중단**
  - 모델 저장에 관한 설정은 앞 절에서 사용한 내용을 그대로 따르겠음
  - 다만 이번에는 최고의 모델 하나만 저장되게끔 해 보자
  - 이를 위해 저장될 모델 이름에 에포크나 정확도 정보를 포함하지 않고, ModelCheckpoint()의 save_best_only 옵션을 True로 설정

```
modelpath = "./data/model/Ch14-4-bestmodel.hdf5"

checkpointer = ModelCheckpoint(filepath=modelpath, monitor='val_loss',
verbose = 0, save_best_only=True)
```

4 학습의 자동 중단

- **학습의 자동 중단**
  - 모델을 실행
  - 자동으로 최적의 에포크를 찾아 멈출 예정이므로 epochs는 넉넉하게 설정

```
history = model.fit(X_train, y_train, epochs=2000, batch_size=500,
validation_split=0.25, verbose=1, callbacks=[early_stopping_callback,
checkpointer])
```

4 학습의 자동 중단

- 학습의 자동 중단
  - 앞서 만든 기본 코드에 다음과 같이 새로운 코드를 불러와 덧붙여 실행해 보자

```
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping

# 학습이 언제 자동 중단될지 설정합니다.
early_stopping_callback = EarlyStopping(monitor='val_loss', patience=20)

# 최적화 모델이 저장될 폴더와 모델 이름을 정합니다.
modelpath = "./data/model/Ch14-4-bestmodel.hdf5"

# 최적화 모델을 업데이트하고 저장합니다.
checkpointer = ModelCheckpoint(filepath=modelpath, monitor='val_loss',
verbose=0, save_best_only=True)
```

4 학습의 자동 중단

- 학습의 자동 중단

```
# 모델을 실행합니다.
history = model.fit(X_train, y_train, epochs=2000, batch_size=500,
validation_split=0.25, verbose=1, callbacks=[early_stopping_callback,
checkpointer])
```

4 학습의 자동 중단

- 학습의 자동 중단

실행 결과

```
Epoch 1/2000
8/8 [==============================] - 0s 18ms/step - loss: 21.4771 - accu
racy: 0.2494 - val_loss: 14.8183 - val_accuracy: 0.2462
... (중략) ...
Epoch 394/2000
8/8 [==============================] - 0s 5ms/step - loss: 0.0500 -
accuracy: 0.9828 - val_loss: 0.0651 - val_accuracy: 0.9846
```

- 에포크를 2,000번으로 설정했지만 394번에서 멈추었음
- 이때의 모델이 model 폴더에 Ch14-4-bestmodel.hdf 라는 이름으로 저장된 것을 확인

4 학습의 자동 중단

- **학습의 자동 중단**
  - 이제 지금까지 만든 모델을 테스트해 보자
  - 따로 저장되어 학습 과정에 포함되지 않은 X_test와 y_test에 지금의 모델을 적용한 결과는 다음과 같음

```
score = model.evaluate(X_test, y_test)
print('Test accuracy:', score[1])
```

4 학습의 자동 중단

- 학습의 자동 중단

실행 결과

```
41/41 [==============================] - 0s 1ms/step - loss: 0.0472 -
accuracy: 0.9885
Test accuracy: 0.9884615540504456
```

- 정확도가 98.84%
- 14.1절에서 실행했던 기본 소스가 94.15%의 정확도를 보였던 것과 비교하면 모델 성능이 대폭 향상된 것을 알 수 있음