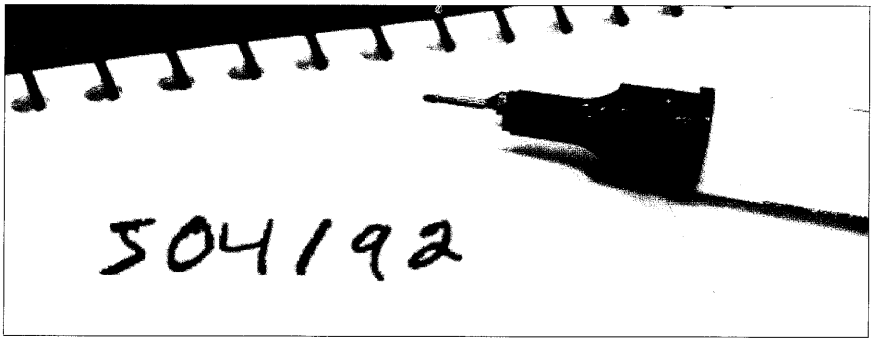
16장 이미지 인식의 꽃, CNN 익히기

DEEP LEARNING FOR EVERYONE

실습과제 MNIST 손글씨 인식



급히 전달받은 노트에 숫자가 적혀 있습니다. 뭐라고 썼는지 읽기에 그리 어렵지 않습니다. 일반적인 사람에게 이 사진에 나온 숫자를 읽어보라 하면 대부분 '504192'라고 읽겠지요.

그런데 컴퓨터에게 이 글씨를 읽게 하고 이 글씨가 어떤 의미인지를 알게 하는 과정은 쉽지 않습니다. 사람이 볼 때는 쉽게 알 수 있는 글씨라 해도, 숫자 5는 어떤 특징을 가졌고, 숫자 9는 6과 어떻게 다른지를 기계가 스스로 파악하여 정확하게 읽고 판단하게 만드는 것은 머신러닝의 오랜 진입 과제였습니다.

MNIST 데이터셋은 미국 국립표준기술원(NIST)이 고등학생과 인구조사국 직원 등이 쓴 손글씨를 이용해 만든 데이터로 구성되어 있습니다. 70,000개의 글자 이미지에 각각 0부터 9까지 이름표를 붙인 데이터셋으로, 머신러닝을 배우는 사람이라면 자신의 알고리즘과 다른 알고리즘의 성과를 비교해 보고자 한 번씩 도전해 보는 가장 유명한 데이터 중 하나이지요.

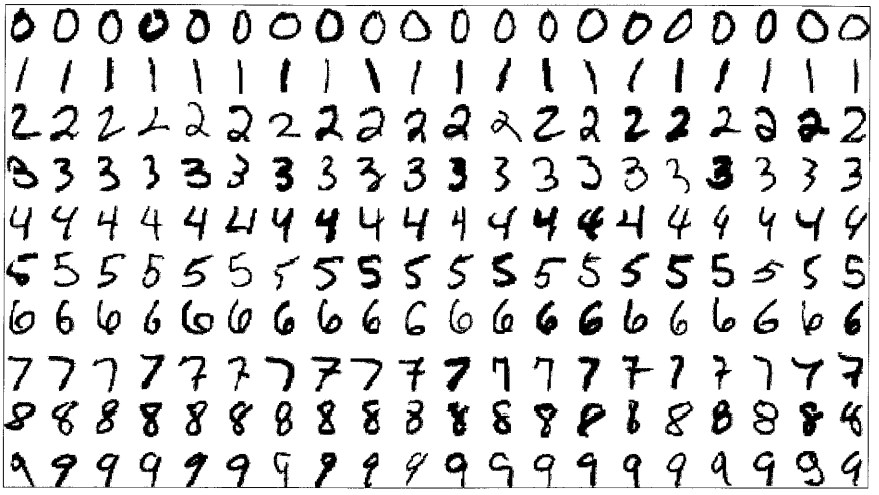


그림 16-1
MNIST 손글씨 데이터 이미지

지금까지 배운 딥러닝을 이용해 과연 이 손글씨 이미지를 몇 %나 정확히 예측할 수 있을까요?

1 데이터 전처리

MNIST 데이터는 케라스를 이용해 간단히 불러올 수 있습니다. `mnist.load_data()` 함수로 사용할 데이터를 불러옵니다.

```
from keras.datasets import mnist
```

이때 불러온 이미지 데이터를 X로, 이 이미지에 0~9까지 붙인 이름표를 Y_class로 구분하여 명명하겠습니다. 또한, 70,000개 중 학습에 사용될 부분은 train으로, 테스트에 사용될 부분은 test라는 이름으로 불러오겠습니다.

- 학습에 사용될 부분: X_train, Y_class_train
- 테스트에 사용될 부분: X_test, Y_class_test

```
(X_train, Y_class_train), (X_test, Y_class_test) = mnist.load_data()
```

케라스의 MNIST 데이터는 총 70,000개의 이미지 중 60,000개를 학습용으로, 10,000개를 테스트용으로 미리 구분해 놓고 있습니다. 이를 다음과 같이 확인할 수 있습니다.

```
print("학습셋 이미지 수: %d 개" % (X_train.shape[0]))
print("테스트셋 이미지 수: %d 개" % (X_test.shape[0]))
```

```
학습셋 이미지 수: 60000 개
테스트셋 이미지 수: 10000 개
```

불러온 이미지 중 한 개만 다시 불러와 보겠습니다. 이를 위해 먼저 matplotlib 라이브러리를 불러옵니다. 그리고 imshow( ) 함수를 이용해 이미지를 출력할 수 있습니다. 모든 이미지가 X_train에 저장되어 있으므로 X_train[0]을 통해 첫 번째 이미지를, cmap = 'Greys' 옵션을 지정해 흑백으로 출력되게 합니다.

```
import matplotlib.pyplot as plt
plt.imshow(X_train[0], cmap='Greys')
plt.show()
```

실행하면 그림 16-2와 같이 이미지가 출력됩니다.

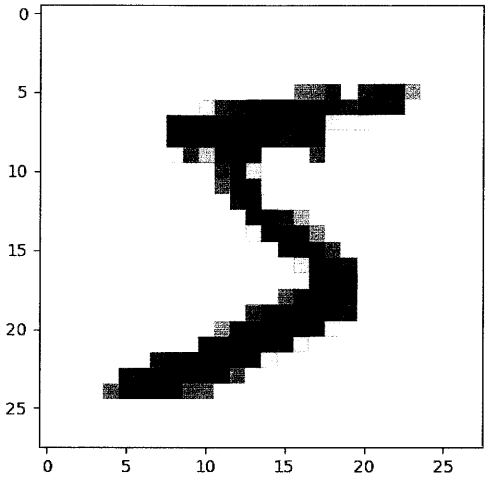


그림 16-2
MNIST 손글씨
데이터의 첫 번째 이미지

이 이미지를 컴퓨터는 어떻게 인식할까요?

이 이미지는 가로 28 × 세로 28 = 총 784개의 픽셀로 이루어져 있습니다. 각 픽셀은 밝기 정도에 따라 0부터 255까지의 등급을 매깁니다. 흰색 배경이 0이라면 글씨가 들어간 곳은 1~255까지 숫자 중 하나로 채워져 긴 행렬로 이루어진 하나의 집합으로 변환됩니다.

다음 코드로 이를 확인할 수 있습니다.

```
for x in X_train[0]:
    for i in x:
        sys.stdout.write('%d\t' % i)
    sys.stdout.write('\n')
```

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	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	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	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	3	18	18	18	126	136	175	26	166	255	247	127	0	0	0	0
0	0	0	0	0	0	0	0	30	36	94	154	170	253	253	253	253	253	225	172	253	242	195	64	0	0	0	0
0	0	0	0	0	0	0	49	238	253	253	253	253	253	253	253	253	251	93	82	82	56	39	0	0	0	0	0
0	0	0	0	0	0	0	18	219	253	253	253	253	253	198	182	247	241	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	80	156	107	253	253	205	11	0	43	154	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	14	1	154	253	90	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	139	253	190	2	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	11	190	253	70	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	35	241	225	160	108	1	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	81	240	253	253	119	25	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	45	186	253	253	150	27	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	16	93	252	253	187	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	249	253	249	64	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	46	130	183	253	253	207	2	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	39	148	229	253	253	253	250	182	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	24	114	221	253	253	253	253	201	78	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	23	66	213	253	253	253	253	198	81	2	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	18	171	219	253	253	253	253	195	80	9	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	55	172	226	253	253	253	253	244	133	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	136	253	253	253	212	135	132	16	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	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	0	0	0	0	0	0	0	0	0	0	0	0	0	0

그림 16-3
그림의 각 좌표를 숫자로 표현해보기

바로 이렇게 이미지는 다시 숫자의 집합으로 바뀌어 학습셋으로 사용됩니다. 우리가 앞서 배운 여러 예제와 마찬가지로 속성을 담은 데이터를 딥러닝에 집어넣고 클래스를 예측하는 문제로 전환시키는 것이지요. 28 × 28 = 784개의 속성을 이용해 0~9까지 10개 클래스 중 하나를 맞히는 문제가 됩니다.

이제 주어진 가로 28, 세로 28의 2차원 배열을 784개의 1차원 배열로 바꿔 주어야 합니다. 이를 위해 reshape( ) 함수를 사용합니다.

reshape(총 샘플 수, 1차원 속성의 수) 형식으로 지정합니다. 총 샘플 수는 앞서 사용한 X_train.shape[0]을 이용하고, 1차원 속성의 수는 이미 살펴본 대로 784개입니다.

```
X_train = X_train.reshape(X_train.shape[0], 784)
```

케라스는 데이터를 0에서 1 사이의 값으로 변환한 다음 구동할 때 최적의 성능을 보입니다. 따라서 현재 0~255 사이의 값으로 이루어진 값을 0~1 사이의 값으로 바꿔야 합니다. 바꾸는 방법은 각 값을 255로 나누는 것입니다. 이렇게 데이터의 폭이 클 때 적절한 값으로 분산의 정도를 바꾸는 과정을 데이터 정규화(normalization)라고 합니다.

현재 주어진 데이터의 값은 0부터 255까지의 정수로, 정규화를 위해 255로 나누어 주려면 먼저 이 값을 실수형으로 바꿔야 합니다. 따라서 다음과 같이 astype( ) 함수를 이용해 실수형으로 바꾼 뒤 255로 나눕니다.

```
X_train = X_train.astype('float64')
X_train = X_train / 255
```

X_test에도 마찬가지로 이 작업을 적용합니다. 다음과 같이 한 번에 적용시키겠습니다.

```
X_test = X_test.reshape(X_test.shape[0], 784).astype('float64') / 255
```

이제 숫자 이미지에 매겨진 이름을 확인해 보겠습니다. 우리는 앞서 불러온 숫자 이미지가 5라는 것을 눈으로 보아 짐작할 수 있습니다. 실제로 이 숫자의 레이블이 어떤지를 불러오고자 Y_class_train[0]을 다음과 같이 출력해 보겠습니다.

```
print("class : %d " % (Y_class_train[0]))
```

그러면 이 숫자의 레이블 값인 5가 출력되는 것을 볼 수 있습니다.

```
class : 5
```

그런데 12장에서 아이리스 품종을 예측할 때 딥러닝의 분류 문제를 해결하려면 원-핫 인코딩 방식을 적용해야 한다고 배웠습니다(153쪽 참조). 즉, 0~9까지의 정수형 값을 갖는 현재 형태에서 0 또는 1로만 이루어진 벡터로 값을 수정해야 합니다.

지금 우리가 열어본 이미지의 class는 [5]였습니다. 이를 [0,0,0,0,0,1,0,0,0,0]로 바꿔야 합니다. 이를 가능하게 해 주는 함수가 바로 np_utils.to_categorical( ) 함수입니다. to_categorical(클래스, 클래스의 개수)의 형식으로 지정합니다.

```
Y_train = np_utils.to_categorical(Y_class_train,10)
Y_test = np_utils.to_categorical(Y_class_test,10)
```

이제 변환된 값을 출력해 보겠습니다.

```
print(Y_train[0])
```

아래와 같이 원-핫 인코딩이 적용된 것을 확인할 수 있습니다.

```
[ 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
```

이제 딥러닝을 실행할 준비를 모두 마쳤습니다.

MNIST 손글씨 인식하기: 데이터 전처리

코드 16-1

예제 소스 run_project/14_MNIST_Data.ipynb

```
from keras.datasets import mnist
from keras.utils import np_utils

import numpy
import sys
import tensorflow as tf

# seed 값 설정
seed = 0
numpy.random.seed(seed)
tf.random.set_seed(3)

# MNIST 데이터셋 불러오기
(X_train, Y_class_train), (X_test, Y_class_test) = mnist.load_
data( )

print("학습셋 이미지 수 : %d 개" % (X_train.shape[0]))
print("테스트셋 이미지 수 : %d 개" % (X_test.shape[0]))

# 그래프로 확인
import matplotlib.pyplot as plt
plt.imshow(X_train[0], cmap='Greys')
plt.show( )

# 코드로 확인
for x in X_train[0]:
```

```
    for i in x:
        sys.stdout.write('%d\t' % i)
    sys.stdout.write('\n')

# 차원 변환 과정
X_train = X_train.reshape(X_train.shape[0], 784)
X_train = X_train.astype('float64')
X_train = X_train / 255

X_test = X_test.reshape(X_test.shape[0], 784).astype('float64') /
255

# 클래스 값 확인
print("class : %d " % (Y_class_train[0]))

# 바이너리화 과정
Y_train = np_utils.to_categorical(Y_class_train, 10)
Y_test = np_utils.to_categorical(Y_class_test, 10)

print(Y_train[0])
```

2 딥러닝 기본 프레임 만들기

이제 불러온 데이터를 실행할 차례입니다. 총 60,000개의 학습셋과 10,000개의 테스트셋을 불러와 속성 값을 지닌 X, 클래스 값을 지닌 Y로 구분하는 작업을 다시 한번 정리하면 다음과 같습니다.

```
from keras.datasets import mnist

(X_train, Y_train), (X_test, Y_test) = mnist.load_data()

X_train = X_train.reshape(X_train.shape[0], 784).
astype('float32') / 255
X_test = X_test.reshape(X_test.shape[0], 784).astype('float32') /
255

Y_train = np_utils.to_categorical(Y_train, 10)
Y_test = np_utils.to_categorical(Y_test, 10)
```

이제 딥러닝을 실행하고자 프레임을 설정합니다. 총 784개의 속성이 있고 10개의 클래스가 있습니다. 따라서 다음과 같이 딥러닝 프레임을 만들 수 있습니다.

```
model = Sequential()
model.add(Dense(512, input_dim=784, activation='relu'))
model.add(Dense(10, activation='softmax'))
```

입력 값(input_dim)이 784개, 은닉층이 512개 그리고 출력이 10개인 모델입니다. 활성화 함수로 은닉층에서는 relu를, 출력층에서는 softmax를 사용했습니다. 그리고 딥러닝 실행 환경을 위해 오차 함수로 categorical_crossentropy, 최적화 함수로 adam을 사용하겠습니다.

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

모델의 실행에 앞서 모델의 성과를 저장하고 모델의 최적화 단계에서 학습을 자동 중단하게끔 설정하겠습니다. 14장에서 배운 내용과 같습니다(195쪽 참조). 10회 이상 모델의 성과 향상이 없으면 자동으로 학습을 중단합니다.

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

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

modelpath="./model/{epoch:02d}-{val_loss:.4f}.hdf5"
checkpointer = ModelCheckpoint(filepath=modelpath, monitor='val_
loss', verbose=1, save_best_only=True)
early_stopping_callback = EarlyStopping(monitor='val_loss',
patience=10)
```

샘플 200개를 모두 30번 실행하게끔 설정합니다. 그리고 테스트셋으로 최종 모델의 성과를 측정하여 그 값을 출력합니다.

```
history = model.fit(X_train, Y_train, validation_data=(X_test, Y_
test), epochs=30, batch_size=200, verbose=0, callbacks=[early_
stopping_callback,checkpointer])

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

실행 결과를 그래프로 표현해 보겠습니다. 역시 14장에서 실습한 내용과 크게 다르지 않습니다. 다만 이번에는 학습셋의 정확도 대신 학습셋의 오차를 그래프로 표현하겠습니다. 학습셋의 오차는 1에서 학습셋의 정확도를 뺀 값입니다. 좀 더 세밀한 변화를 볼 수 있게 학습셋의 오차와 테스트셋의 오차를 그래프 하나로 나타내겠습니다.

```
import matplotlib.pyplot as plt

y_vloss = history.history['val_loss']

# 학습셋의 오차
y_loss = history.history['loss']

# 그래프로 표현
x_len = numpy.arange(len(y_loss))
plt.plot(x_len, y_vloss, marker='.', c="red", label='Testset_
loss')
plt.plot(x_len, y_loss, marker='.', c="blue", label='Trainset_
loss')

# 그래프에 그리드를 주고 레이블을 표시
plt.legend(loc='upper right')
plt.grid()
plt.xlabel('epoch')
plt.ylabel('loss')
plt.show()
```

지금까지 내용을 한 스크립트로 정리하면 다음과 같습니다.

MNIST 손글씨 인식하기: 기본 프레임

코드 16-2

예제 소스 run_project/15_MNIST_Simple.ipynb

```
from keras.datasets import mnist
from keras.utils import np_utils
from keras.models import Sequential
from keras.layers import Dense
from keras.callbacks import ModelCheckpoint,EarlyStopping

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

# seed 값 설정
seed = 0
numpy.random.seed(seed)
tf.set_random_seed(3)

# MNIST 데이터 불러오기
(X_train, Y_train), (X_test, Y_test) = mnist.load_data()

X_train = X_train.reshape(X_train.shape[0], 784).
astype('float32') / 255
X_test = X_test.reshape(X_test.shape[0], 784).astype('float32') /
255

Y_train = np_utils.to_categorical(Y_train, 10)
Y_test = np_utils.to_categorical(Y_test, 10)
```

```
# 모델 프레임 설정
model = Sequential()
model.add(Dense(512, input_dim=784, activation='relu'))
model.add(Dense(10, activation='softmax'))

# 모델 실행 환경 설정
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics=['accuracy'])

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

modelpath="./model/{epoch:02d}-{val_loss:.4f}.hdf5"
checkpointer = ModelCheckpoint(filepath=modelpath, monitor='val_
loss', verbose=1, save_best_only=True)
early_stopping_callback = EarlyStopping(monitor='val_loss',
patience=10)

# 모델의 실행
history = model.fit(X_train, Y_train, validation_data=(X_test, Y_
test), epochs=30, batch_size=200, verbose=0, callbacks=[early_
stopping_callback,checkpointer])

# 테스트 정확도 출력
print("\n Test Accuracy: %.4f" % (model.evaluate(X_test, Y_test)
[1]))

# 테스트셋의 오차
y_vloss = history.history['val_loss']
```

```
# 학습셋의 오차
y_loss = history.history['loss']

# 그래프로 표현
x_len = numpy.arange(len(y_loss))
plt.plot(x_len, y_vloss, marker='.', c="red", label='Testset_
loss')
plt.plot(x_len, y_loss, marker='.', c="blue", label='Trainset_
loss')

# 그래프에 그리드를 주고 레이블을 표시
plt.legend(loc='upper right')
# plt.axis([0, 20, 0, 0.35])
plt.grid()
plt.xlabel('epoch')
plt.ylabel('loss')
plt.show()
```

실행 결과

```
Epoch 00000: val_loss improved from inf to 0.15460, saving model to ./
model/00-0.1546.hdf5
Epoch 00001: val_loss improved from 0.15460 to 0.10492, saving model to ./
model/01-0.1049.hdf5
Epoch 00002: val_loss improved from 0.10492 to 0.08447, saving model to ./
model/02-0.0845.hdf5
Epoch 00003: val_loss improved from 0.08447 to 0.07896, saving model to ./
model/03-0.0790.hdf5
Epoch 00004: val_loss improved from 0.07896 to 0.06699, saving model to ./
model/04-0.0670.hdf5
Epoch 00005: val_loss improved from 0.06699 to 0.06388, saving model to ./
model/05-0.0639.hdf5
```

```
Epoch 00006: val_loss did not improve
Epoch 00007: val_loss improved from 0.06388 to 0.06291, saving model to ./
model/07-0.0629.hdf5
Epoch 00008: val_loss improved from 0.06291 to 0.05828, saving model to ./
model/08-0.0583.hdf5
Epoch 00009: val_loss did not improve
Epoch 00010: val_loss did not improve
Epoch 00011: val_loss did not improve
Epoch 00012: val_loss did not improve
Epoch 00013: val_loss did not improve
Epoch 00014: val_loss did not improve
Epoch 00015: val_loss did not improve
Epoch 00016: val_loss did not improve
Epoch 00017: val_loss did not improve
Epoch 00018: val_loss did not improve
Epoch 00019: val_loss did not improve
 9920/10000 [============================>.] - ETA: 0s
 Test Accuracy: 0.9821
```

20번째 실행에서 멈춘 것을 확인할 수 있습니다. 베스트 모델은 10번째 에포크일 때이며, 이 모델의 테스트셋에 대한 정확도는 98.21%입니다. 함께 출력되는 그래프로 실행 내용을 확인할 수 있습니다.

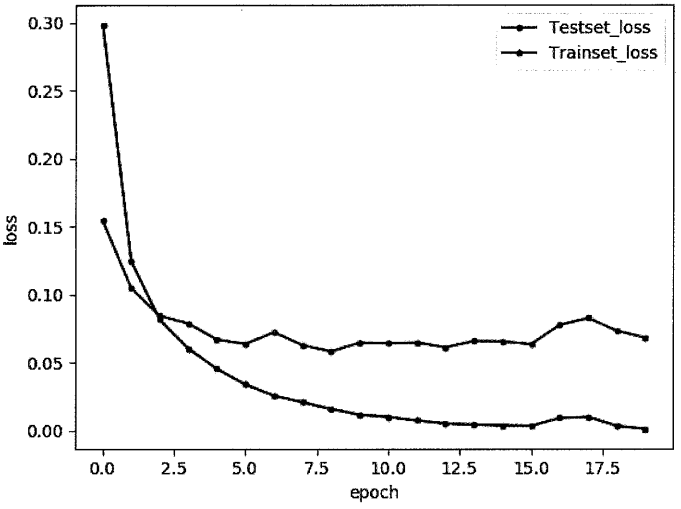


그림 16-4
학습이 진행될 때
학습셋과
테스트셋의 오차 변화

학습셋에 대한 오차는 계속해서 줄어듭니다. 테스트셋의 과적합이 일어나기 전 학습을 끝낸 모습입니다.

TIP 그림 16-4는 학습셋의 오차(Trainset_loss)와 테스트셋의 오차(Testset_loss)를 그래프로 표현한 것입니다. 앞서 학습셋과 테스트셋을 배울 때는 학습셋의 정확도(Trainset_acc)를 이용했습니다(그림 14-1). 하지만 학습셋의 정확도는 1.00에 가깝고 테스트셋의 오차는 0.00에 가까우므로 두 개를 함께 비교하기가 어렵습니다. 따라서 1에서 학습셋의 정확도를 뺀 값, 즉 학습셋의 오차를 주로 그래프에 적용하여 이와 같이 표현합니다.

3 더 깊은 딥러닝

앞서 98.21%의 정확도를 보인 딥러닝 프레임은 하나의 은닉층을 둔 아주 단순한 모델입니다. 이를 도식화해서 표현하면 그림 16-5와 같습니다.

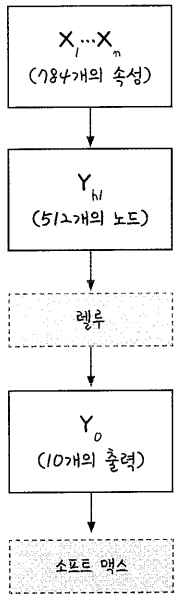


그림 16-5
은닉층이 하나인 딥러닝 모델의 도식

딥러닝은 이러한 기본 모델을 바탕으로, 프로젝트에 맞춰서 어떤 옵션을 더하고 어떤 층을 추가하느냐에 따라 성능이 좋아질 수 있습니다. 지금부터는 기본 딥러닝 프레임에 이미지 인식 분야에서 강력한 성능을 보이는 컨볼루션 신경망(Convolutional Neural Network, CNN)을 얹어보겠습니다.

4 컨볼루션 신경망(CNN)

컨볼루션 신경망은 입력된 이미지에서 다시 한번 특징을 추출하기 위해 마스크(필터, 윈도 또는 커널이라고도 함)를 도입하는 기법입니다. 예를 들어, 입력된 이미지가 다음과 같은 값을 가지고 있다고 합시다.

1	0	1	0
0	1	1	0
0	0	1	1
0	0	1	0

여기에 2×2 마스크를 준비합니다. 각 칸에는 가중치가 들어있습니다. 샘플 가중치를 다음과 같이 ×1, ×0라고 하겠습니다.

×1	×0
×0	×1

이제 마스크를 맨 왼쪽 위칸에 적용시켜 보겠습니다.

1×1	0×0	1	0
0×0	1×1	1	0
0	0	1	1
0	0	1	0

적용된 부분은 원래 있던 값에 가중치의 값을 곱해 줍니다. 그 결과를 합하면 새로 추출된 값은 2가 됩니다.

$$(1\times1)+(0\times0)+(0\times0)+(1\times1)=2$$

이 마스크를 한 칸씩 옮겨 모두 적용해 보겠습니다.

1×1	0×0	1	0
0×0	1×1	1	0
0	0	1	1
0	0	1	0

1	0×1	1×0	0
0	1×0	1×1	0
0	0	1	1
0	0	1	0

1	0	1×1	0×0
0	1	1×0	0×1
0	0	1	1
0	0	1	0

1	0	1	0
0×1	1×0	1	0
0×0	0×1	1	1
0	0	1	0

1	0	1	0
0	1×1	1×0	0
0	0×0	1×1	1
0	0	1	0

1	0	1	0
0	1	1×1	0×0
0	0	1×0	1×1
0	0	1	0

1	0	1	0
0	1	1	0
0×1	0×0	1	1
0×0	0×1	1	0

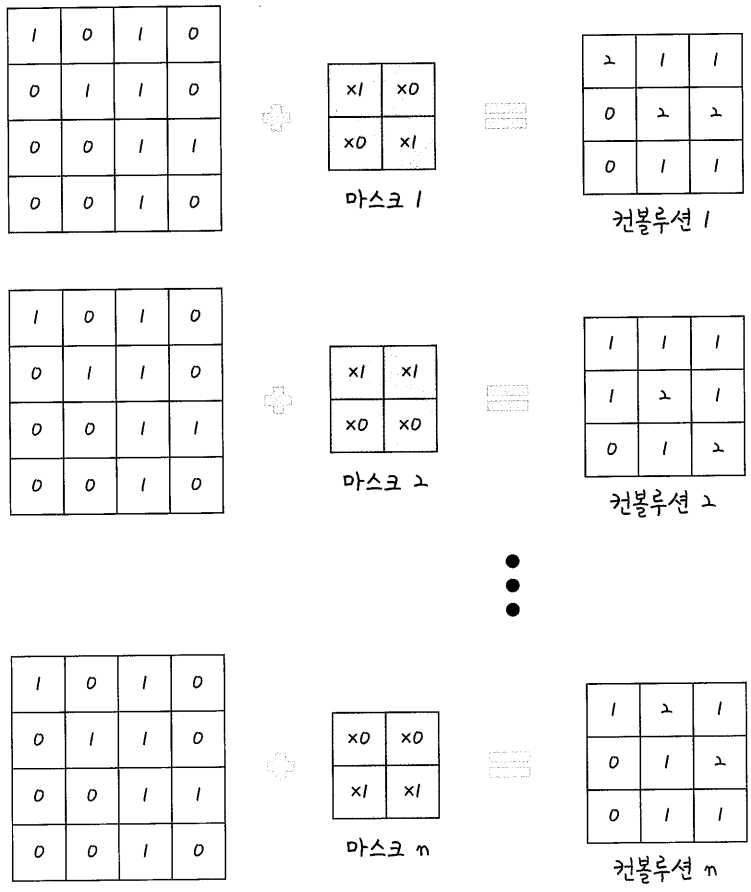
1	0	1	0
0	1	1	0
0	0×1	1×0	1
0	0×0	1×1	0

1	0	1	0
0	1	1	0
0	0	1×1	1×0
0	0	1×0	0×1

그 결과를 정리하면 다음과 같습니다.

2	1	1
0	2	2
0	1	1

이렇게 해서 새롭게 만들어진 층을 컨볼루션(합성곱)이라고 부릅니다. 컨볼루션을 만들면 입력 데이터로부터 더욱 정교한 특징을 추출할 수 있습니다. 이러한 마스크를 여러 개 만들 경우 여러 개의 컨볼루션이 만들어집니다.



케라스에서 컨볼루션 층을 추가하는 함수는 Conv2D( )입니다. 다음과 같이 컨볼루션 층을 적용하여 MNIST 손글씨 인식률을 높여봅시다.

```
model.add(Conv2D(32, kernel_size=(3, 3), input_shape=(28, 28, 1),
activation='relu'))
```

여기에 입력된 4가지 인자는 다음과 같습니다.

1 | 첫 번째 인자: 마스크를 몇 개 적용할지 정합니다. 앞서 살펴본 것처럼 여러 개의 마스크를 적용하면 서로 다른 컨볼루션이 여러 개 나옵니다. 여기서는 32개의 마스크를 적용했습니다.

2 | kernel_size: 마스크(커널)의 크기를 정합니다. kernel_size=(행, 열) 형식으로 정하며, 여기서는 3×3 크기의 마스크를 사용하게끔 정하였습니다.

3 | input_shape: Dense 층과 마찬가지로 맨 처음 층에는 입력되는 값을 알려주어야 합니다. input_shape=(행, 열, 색상 또는 흑백) 형식으로 정합니다. 만약 입력 이미지가 색상이면 3, 흑백이면 1을 지정합니다.

4 | activation: 활성화 함수를 정의합니다.

컨볼루션 층을 하나 더 추가하겠습니다. 다음과 같이 마스크 64개를 적용한 새로운 컨볼루션 층을 추가할 수 있습니다.

```
model.add(Conv2D(64, (3, 3), activation='relu'))
```

컨볼루션 층을 추가한 도식을 그려보면 그림 16-6과 같습니다.

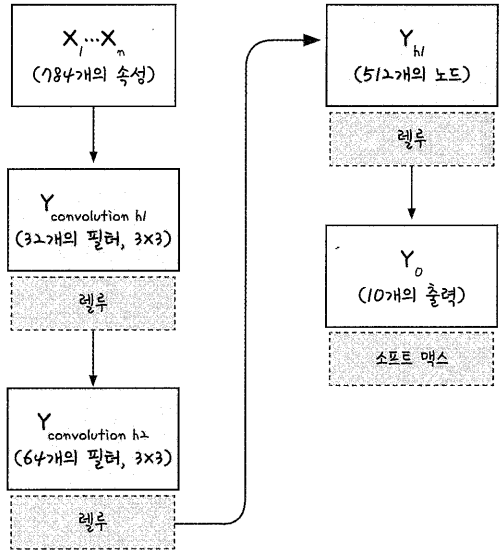
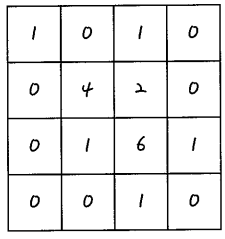


그림 16-6
컨볼루션 층의 적용

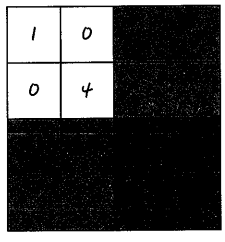
5 맥스 풀링

앞서 구현한 컨볼루션 층을 통해 이미지 특징을 도출하였습니다. 하지만 그 결과가 여전히 크고 복잡하면 이를 다시 한번 축소해야 합니다. 이 과정을 풀링(pooling) 또는 서브 샘플링(sub sampling)이라고 합니다.

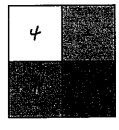
이러한 풀링 기법에는 정해진 구역 안에서 최댓값을 뽑아내는 **맥스 풀링**(max pooling)과 평균값을 뽑아내는 **평균 풀링**(average pooling) 등이 있습니다. 이중 보편적으로 사용되는 맥스 풀링의 예를 들어 보겠습니다. 다음과 같은 이미지가 있다고 합시다.



맥스 풀링을 적용하면 다음과 같이 구역을 나눕니다.



그리고 각 구역에서 가장 큰 값을 추출합니다.



이 과정을 거쳐 불필요한 정보를 간추립니다. 맥스 풀링은 `MaxPooling2D( )` 함수를 사용해서 다음과 같이 적용할 수 있습니다.

```
model.add(MaxPooling2D(pool_size=2))
```

여기서 pool_size는 풀링 창의 크기를 정하는 것으로, 2로 정하면 전체 크기가 절반으로 줄어듭니다.

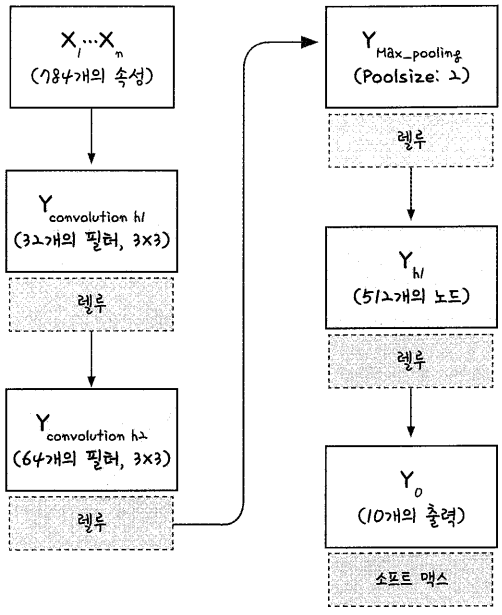


그림 16-7
맥스 풀링 층 추가

■ 드롭아웃, 플래튼

노드가 많아지거나 층이 많아진다고 해서 학습이 무조건 좋아지는 것이 아니라는 점을 과적합의 의미를 공부하며 배웠습니다. 딥러닝 학습을 실행할 때 가장 중요한 것은 과적합을 얼마나 효과적으로 피해가는지에 달려 있다고 해도 과언이 아닙니다. 따라서 그동안 이러한 과정을 도와주는 기법이 연구되어 왔습니다. 그중 간단하지만 효과가 큰 기법이 바로 **드롭아웃**(drop out) 기법입니다. 드롭아웃은 은닉층에 배치된 노드 중 일부를 임의로 꺼주는 것입니다.

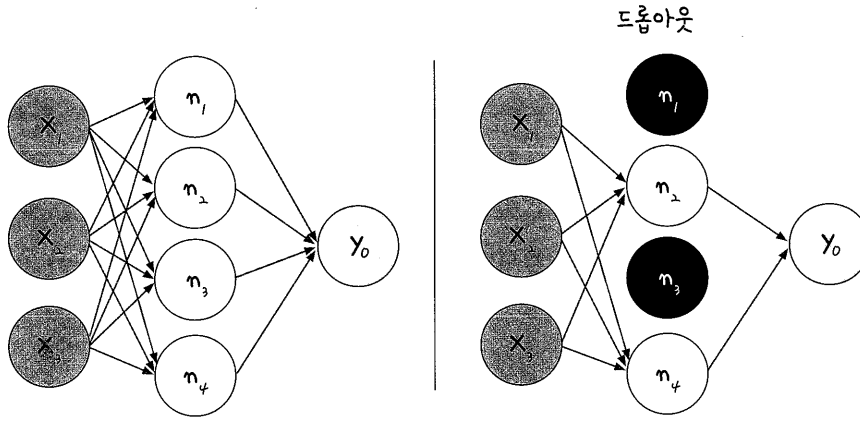


그림 16-8
드롭아웃의 개요, 검은색으로 표시된 노드는 계산하지 않는다.

이렇게 랜덤하게 노드를 끔으로써 학습 데이터에 지나치게 치우쳐서 학습되는 과적합을 방지할 수 있습니다. 케라스는 이를 손쉽게 적용하도록 도와줍니다. 예를 들어, 25%의 노드를 끄려면 다음과 같이 코드를 작성해 줍니다.

```
model.add(Dropout(0.25))
```

이제 이러한 과정을 지나 다시 앞에서 Dense( ) 함수를 이용해 만들었던 기본 층에 연결해 볼까요? 이때 주의할 점은 컨볼루션 층이나 맥스 풀링은 주어진 이미지를 2차원 배열인 채로 다룬다는 점입니다. 이를 1차원 배열로 바꿔주어야 활성화 함수가 있는 층에서 사용할 수 있습니다. 따라서 Flatten( ) 함수를 사용해 2차원 배열을 1차원으로 바꿔줍니다.

```
model.add(Flatten())
```

이를 포함하여 새롭게 구현할 딥러닝 프레임을 그림 16-9와 같이 설정해 보겠습니다.

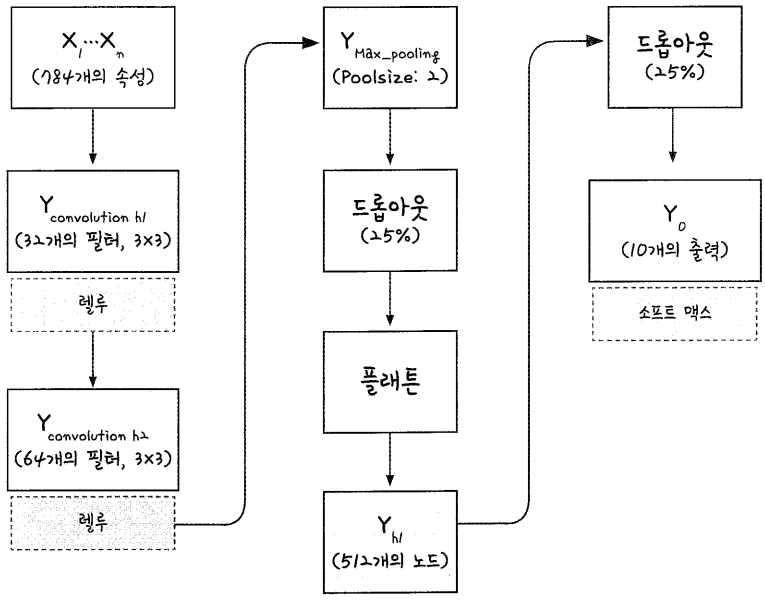


그림 16-9
드롭아웃과 플래튼 추가하기

6 컨볼루션 신경망 실행하기

지금까지 살펴본 내용을 코드로 작성해 보겠습니다. 앞서 코드 16-2에서 만든 딥러닝 기본 프레임을 그대로 이용하되 model 설정 부분만 지금까지 나온 내용으로 바꿔주면 됩니다.

MNIST 손글씨 인식하기: 컨볼루션 신경망 적용 코드 16-3

예제 소스 run_project/16_MNIST_Deep.ipynb

```
from keras.datasets import mnist
from keras.utils import np_utils
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten, Conv2D,
```

```
MaxPooling2D
from keras.callbacks import ModelCheckpoint,EarlyStopping

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

# seed 값 설정
seed = 0
numpy.random.seed(seed)
tf.random.set_seed(3)

# 데이터 불러오기
(X_train, Y_train), (X_test, Y_test) = mnist.load_data()
X_train = X_train.reshape(X_train.shape[0], 28, 28,
1).astype('float32') / 255
X_test = X_test.reshape(X_test.shape[0], 28, 28,
1).astype('float32') / 255
Y_train = np_utils.to_categorical(Y_train)
Y_test = np_utils.to_categorical(Y_test)

# 컨볼루션 신경망 설정
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), input_shape=(28, 28, 1),
activation='relu'))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=2))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
```

```
model.add(Dense(10, activation='softmax'))

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

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

modelpath="./model/{epoch:02d}-{val_loss:.4f}.hdf5"
checkpointer = ModelCheckpoint(filepath=modelpath, monitor='val_
loss', verbose=1, save_best_only=True)
early_stopping_callback = EarlyStopping(monitor='val_loss',
patience=10)

# 모델의 실행
history = model.fit(X_train, Y_train, validation_data=(X_test, Y_
test), epochs=30, batch_size=200, verbose=0, callbacks=[early_
stopping_callback,checkpointer])

# 테스트 정확도 출력
print("\n Test Accuracy: %.4f" % (model.evaluate(X_test, Y_test)
[1]))

# 테스트셋의 오차
y_vloss = history.history['val_loss']

# 학습셋의 오차
y_loss = history.history['loss']
```

```
# 그래프로 표현
x_len = numpy.arange(len(y_loss))
plt.plot(x_len, y_vloss, marker='.', c="red", label='Testset_
loss')
plt.plot(x_len, y_loss, marker='.', c="blue", label='Trainset_
loss')

# 그래프에 그리드를 주고 레이블을 표시
plt.legend(loc='upper right')
plt.grid()
plt.xlabel('epoch')
plt.ylabel('loss')
plt.show()
```

실행 결과

```
Epoch 00000: val_loss improved from inf to 0.06068, saving model to ./
model/00-0.0607.hdf5
Epoch 00001: val_loss improved from 0.06068 to 0.04409, saving model to ./
model/01-0.0441.hdf5
Epoch 00002: val_loss improved from 0.04409 to 0.03909, saving model to ./
model/02-0.0391.hdf5
Epoch 00003: val_loss improved from 0.03909 to 0.03188, saving model to ./
model/03-0.0319.hdf5
Epoch 00004: val_loss improved from 0.03188 to 0.02873, saving model to ./
model/04-0.0287.hdf5
Epoch 00005: val_loss did not improve
Epoch 00006: val_loss did not improve
Epoch 00007: val_loss did not improve
Epoch 00008: val_loss improved from 0.02873 to 0.02678, saving model to ./
model/08-0.0268.hdf5
```

```
Epoch 00009: val_loss did not improve
Epoch 00010: val_loss improved from 0.02678 to 0.02617, saving model to ./model/10-0.0262.hdf5
Epoch 00011: val_loss did not improve
Epoch 00012: val_loss did not improve
Epoch 00013: val_loss improved from 0.02617 to 0.02454, saving model to ./model/13-0.0245.hdf5
Epoch 00014: val_loss did not improve
Epoch 00015: val_loss did not improve
Epoch 00016: val_loss did not improve
Epoch 00017: val_loss did not improve
Epoch 00018: val_loss did not improve
Epoch 00019: val_loss did not improve
Epoch 00020: val_loss did not improve
Epoch 00021: val_loss did not improve
Epoch 00022: val_loss did not improve
Epoch 00023: val_loss did not improve
Epoch 00024: val_loss did not improve

Test Accuracy: 0.9928
```

TIP CPU 기반으로 코드 16-3을 실행하면 앞서 다룬 예제보다 시간이 더 오래 걸릴 것입니다. 이 책에서 소개한 작은 프로젝트 정도면 CPU에서 실습해도 큰 상관이 없지만, 더 큰 딥러닝 프로젝트는 반드시 GPU 환경을 갖추어 시행할 것을 추천합니다. 또한, 결과는 실행할 때마다 다를 수도 있습니다.

14번째 에포크에서 베스트 모델을 만들었고 25번째 에포크에서 학습이 자동 중단되었습니다. 테스트 정확도가 99.28%로 향상되었습니다. 다음과 같이 함께 생성된 그래프로 학습 과정을 시각화할 수 있습니다.

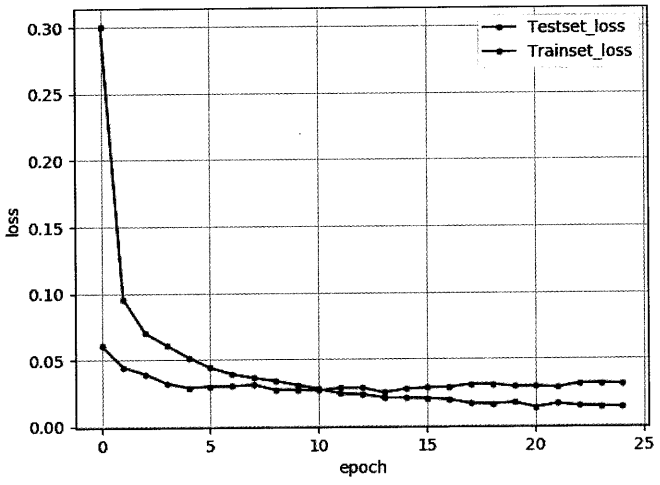


그림 16-10
학습의 진행에 따른 학습셋과 테스트셋의 오차 변화

0.9928, 즉 99.28%의 정확도는 10,000개의 테스트 이미지 중 9,928개를 맞혔다는 뜻입니다. 코드 16-2에서는 정확도가 98.21%였으므로 이보다 107개의 정답을 더 맞힌 것입니다. 100% 다 맞히지 못한 이유는 데이터 안에 다음과 같이 확인할 수 없는 글씨가 있었기 때문입니다.

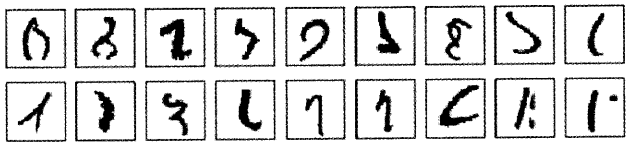


그림 16-11
알아내지 못한 숫자의 예

우리가 만든 딥러닝 모델은 이미 사람의 인식 정도와 같거나 이를 뛰어넘는 인식률을 보여 준다고 해도 과언이 아니지요?