

MNIST 분류 모델 만들기 - 신경망

학습 내용

- 데이터 전처리 후, 딥러닝 모델 돌려보기
 - 데이터 차원 조정
 - 데이터의 값의 범위 변경(0 ~ 255 -> 0 ~ 1)

환경

- google colab
 - tensorflow 2.8.0
 - keras 2.8.0
 - python 3.7.13

목차

[01 라이브러리 임포트 및 데이터 준비](#)

[02 데이터 전처리](#)

[03 딥러닝 모델 구축하기](#)

01 라이브러리 임포트 및 데이터 준비

[목차로 이동하기](#)

In [1]:

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

In [2]:

```
import numpy
import sys
import tensorflow as tf
```

In [3]:

```
# 난수 생성기의 패턴이 지정되지 않았을때,
print( numpy.random.rand(4) )
print( numpy.random.rand(4) )
```

```
[0.58876587 0.23991335 0.72467289 0.63857356]
[0.14204534 0.26480378 0.22780786 0.79339815]
```

In [4]:

```
# 난수 생성기 패턴 지정
numpy.random.seed(0)
print( numpy.random.rand(4) )
numpy.random.seed(0)
print( numpy.random.rand(4) )
```

```
[0.5488135  0.71518937 0.60276338 0.54488318]
[0.5488135  0.71518937 0.60276338 0.54488318]
```

- 난수 패턴기의 패턴이 지정이 되면 같은 난수가 발생된다.

데이터 다운로드

In [5]:

```
# 처음 다운일 경우, 데이터 다운로드 시간이 걸릴 수 있음.
(X_train, y_train), (X_test, y_test) = mnist.load_data()
print(X_train.shape, y_train.shape)
print(X_test.shape, y_test.shape)
```

```
(60000, 28, 28) (60000,)
(10000, 28, 28) (10000,)
```

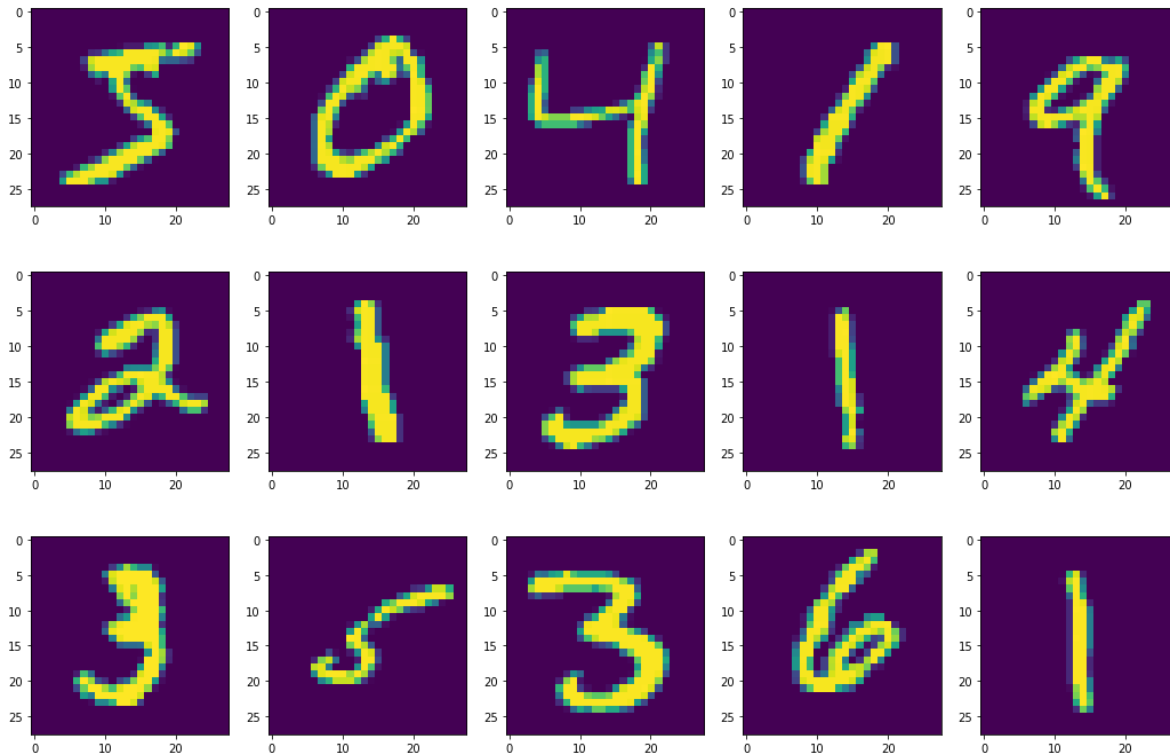
In [6]:

```
import matplotlib.pyplot as plt
```

In [7]:

```
fig, axes = plt.subplots(3, 5, figsize=(18,12) )  
  
print("label={}".format(y_train[0:15])) # x데이터 0~14개 가져오기  
  
for image, ax in zip( X_train, axes.ravel() ):  
    ax.imshow(image) # 이미지 표시
```

label=[5 0 4 1 9 2 1 3 1 4 3 5 3 6 1]



X_train의 데이터 정보를 하나 보기

In [8]:

```
print(X_train.shape) # 60000 만개, 28행, 28열  
X_train[0].shape
```

(60000, 28, 28)

Out[8]:

(28, 28)

02. 데이터 전처리

[목차로 이동하기](#)

신경망에 맞추어 주기 위한 데이터 전처리

- 학습 데이터
- 테스트 데이터

입력 데이터의 텐서형 변경 및 값의 범위 조정

In [9]:

```
X_train = X_train.reshape(X_train.shape[0],784) # 60000, 28, 28 -> 60000, 784로 변경
# 데이터 값의 범위 0~255 -> 0~1
X_train.astype('float64')
X_train = X_train/255

# 한줄 표현 - 이렇게도 가능
# X_train = X_train.reshape(X_train.shape[0],784).astype('float64') / 255
```

In [10]:

```
import numpy as np
```

In [11]:

```
print(X_train.shape) # 데이터 크기
print("데이터의 최대, 최소 :", np.min(X_train), np.max(X_train) ) # 값의 범위
```

```
(60000, 784)
데이터의 최대, 최소 : 0.0 1.0
```

In [12]:

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

In [13]:

```
print(X_test.shape) # 데이터 크기
np.min(X_test), np.max(X_test) # 값의 범위
```

```
(10000, 784)
```

Out[13]:

```
(0.0, 1.0)
```

출력데이터 검증을 위해 10진수의 값을 One-Hot Encoding을 수행

In [14]:

```
# OneHotEncoding - 10진수의 값을 0, 1의 값을 갖는 벡터로 표현
y_train_1D = np_utils.to_categorical(y_train, 10)
y_test_1D = np_utils.to_categorical(y_test, 10)
```

변환 전과 후

In [15]:

```
y_train[0:4]
```

Out[15]:

```
array([5, 0, 4, 1], dtype=uint8)
```

In [16]:

```
y_train_1D[0:4]
```

Out[16]:

```
array([[0., 0., 0., 0., 0., 1., 0., 0., 0., 0.],
       [1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0., 0., 0., 0., 0.]], dtype=float32)
```

03. 딥러닝 모델 구축하기

[목차로 이동하기](#)

In [17]:

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

In [18]:

```
m = Sequential()
m.add(Dense(512, input_dim=784, activation='relu'))
m.add(Dense(128, activation='relu'))
m.add(Dense(10, activation='softmax')) #softmax
```

오차함수 : categorical_crossentropy, 최적화 함수 : adam

In [19]:

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

In [20]:

```
### 배치 사이즈 200, epochs 30회 실행,  
history = m.fit(X_train, y_train_1D, validation_data=(X_test, y_test_1D),  
                epochs=30,  
                batch_size=200,  
                verbose=1)
```

```
Epoch 1/30  
300/300 [=====] - 4s 12ms/step - loss: 0.2652 - accuracy:  
0.9247 - val_loss: 0.1228 - val_accuracy: 0.9615  
Epoch 2/30  
300/300 [=====] - 3s 11ms/step - loss: 0.0961 - accuracy:  
0.9710 - val_loss: 0.0843 - val_accuracy: 0.9738  
Epoch 3/30  
300/300 [=====] - 3s 11ms/step - loss: 0.0625 - accuracy:  
0.9806 - val_loss: 0.0727 - val_accuracy: 0.9777  
Epoch 4/30  
300/300 [=====] - 4s 15ms/step - loss: 0.0430 - accuracy:  
0.9865 - val_loss: 0.0670 - val_accuracy: 0.9808  
Epoch 5/30  
300/300 [=====] - 3s 11ms/step - loss: 0.0307 - accuracy:  
0.9907 - val_loss: 0.0665 - val_accuracy: 0.9800  
Epoch 6/30  
300/300 [=====] - 3s 11ms/step - loss: 0.0235 - accuracy:  
0.9928 - val_loss: 0.0688 - val_accuracy: 0.9799  
Epoch 7/30  
300/300 [=====] - 3s 11ms/step - loss: 0.0184 - accuracy:  
0.9944 - val_loss: 0.0646 - val_accuracy: 0.9828  
Epoch 8/30  
300/300 [=====] - 3s 11ms/step - loss: 0.0142 - accuracy:  
0.9955 - val_loss: 0.0699 - val_accuracy: 0.9804  
Epoch 9/30  
300/300 [=====] - 3s 11ms/step - loss: 0.0109 - accuracy:  
0.9966 - val_loss: 0.0766 - val_accuracy: 0.9798  
Epoch 10/30  
300/300 [=====] - 3s 11ms/step - loss: 0.0137 - accuracy:  
0.9954 - val_loss: 0.0803 - val_accuracy: 0.9780  
Epoch 11/30  
300/300 [=====] - 3s 11ms/step - loss: 0.0104 - accuracy:  
0.9967 - val_loss: 0.0735 - val_accuracy: 0.9821  
Epoch 12/30  
300/300 [=====] - 3s 11ms/step - loss: 0.0092 - accuracy:  
0.9970 - val_loss: 0.0780 - val_accuracy: 0.9798  
Epoch 13/30  
300/300 [=====] - 3s 11ms/step - loss: 0.0108 - accuracy:  
0.9964 - val_loss: 0.0756 - val_accuracy: 0.9813  
Epoch 14/30  
300/300 [=====] - 3s 11ms/step - loss: 0.0075 - accuracy:  
0.9975 - val_loss: 0.0783 - val_accuracy: 0.9818  
Epoch 15/30  
300/300 [=====] - 3s 11ms/step - loss: 0.0059 - accuracy:  
0.9982 - val_loss: 0.0822 - val_accuracy: 0.9806  
Epoch 16/30  
300/300 [=====] - 3s 11ms/step - loss: 0.0053 - accuracy:  
0.9984 - val_loss: 0.1060 - val_accuracy: 0.9767  
Epoch 17/30  
300/300 [=====] - 3s 12ms/step - loss: 0.0056 - accuracy:  
0.9983 - val_loss: 0.0778 - val_accuracy: 0.9836  
Epoch 18/30  
300/300 [=====] - 3s 12ms/step - loss: 0.0054 - accuracy:
```

```

0.9983 - val_loss: 0.0902 - val_accuracy: 0.9804
Epoch 19/30
300/300 [=====] - 3s 11ms/step - loss: 0.0066 - accuracy:
0.9978 - val_loss: 0.0964 - val_accuracy: 0.9794
Epoch 20/30
300/300 [=====] - 3s 11ms/step - loss: 0.0067 - accuracy:
0.9980 - val_loss: 0.0878 - val_accuracy: 0.9805
Epoch 21/30
300/300 [=====] - 3s 11ms/step - loss: 0.0048 - accuracy:
0.9985 - val_loss: 0.0935 - val_accuracy: 0.9811
Epoch 22/30
300/300 [=====] - 3s 11ms/step - loss: 0.0062 - accuracy:
0.9984 - val_loss: 0.0934 - val_accuracy: 0.9825
Epoch 23/30
300/300 [=====] - 3s 11ms/step - loss: 0.0078 - accuracy:
0.9976 - val_loss: 0.0867 - val_accuracy: 0.9825
Epoch 24/30
300/300 [=====] - 3s 11ms/step - loss: 0.0052 - accuracy:
0.9984 - val_loss: 0.0926 - val_accuracy: 0.9829
Epoch 25/30
300/300 [=====] - 3s 11ms/step - loss: 0.0038 - accuracy:
0.9987 - val_loss: 0.0972 - val_accuracy: 0.9821
Epoch 26/30
300/300 [=====] - 3s 11ms/step - loss: 0.0056 - accuracy:
0.9984 - val_loss: 0.0992 - val_accuracy: 0.9811
Epoch 27/30
300/300 [=====] - 3s 11ms/step - loss: 0.0031 - accuracy:
0.9991 - val_loss: 0.0905 - val_accuracy: 0.9833
Epoch 28/30
300/300 [=====] - 3s 11ms/step - loss: 0.0030 - accuracy:
0.9990 - val_loss: 0.0970 - val_accuracy: 0.9823
Epoch 29/30
300/300 [=====] - 3s 11ms/step - loss: 0.0066 - accuracy:
0.9979 - val_loss: 0.1049 - val_accuracy: 0.9820
Epoch 30/30
300/300 [=====] - 3s 11ms/step - loss: 0.0046 - accuracy:
0.9985 - val_loss: 0.1087 - val_accuracy: 0.9807

```

In [21]:

```

print("학습용 데이터 셋 Accuracy : %.4f" %(m.evaluate(X_train, y_train_1D)[1]))
print("테스트용 데이터 셋 Accuracy : %.4f" %(m.evaluate(X_test, y_test_1D)[1]))

```

```

1875/1875 [=====] - 5s 3ms/step - loss: 0.0045 - accuracy:
0.9985
학습용 데이터 셋 Accuracy : 0.9985
313/313 [=====] - 1s 3ms/step - loss: 0.1087 - accuracy: 0.
9807
테스트용 데이터 셋 Accuracy : 0.9807

```

In [22]:

```

pred = m.predict(X_test)

```

In [23]:

```
print( pred.shape )  
print( "예측값 : ", pred[1] )  
print( "예측값 중 가장 높은 값의 위치 : ", np.argmax(pred[1]) )
```

(10000, 10)

예측값 : [9.3413454e-17 1.6887590e-16 1.0000000e+00 4.6736043e-16 1.1535825e-29
6.4255561e-22 1.9321864e-17 7.5378626e-24 9.8010006e-14 9.5771777e-25]

예측값 중 가장 높은 값의 위치 : 2