MNIST 분류 모델 만들기 - 신경망

In [1]:

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

```
Using TensorFlow backend.
```

In [34]:

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

In [35]:

```
seed = 0
numpy.random.seed(seed)
tf.set_random_seed(seed)
```

데이터 다운로드

In [36]:

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

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

In [37]:

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

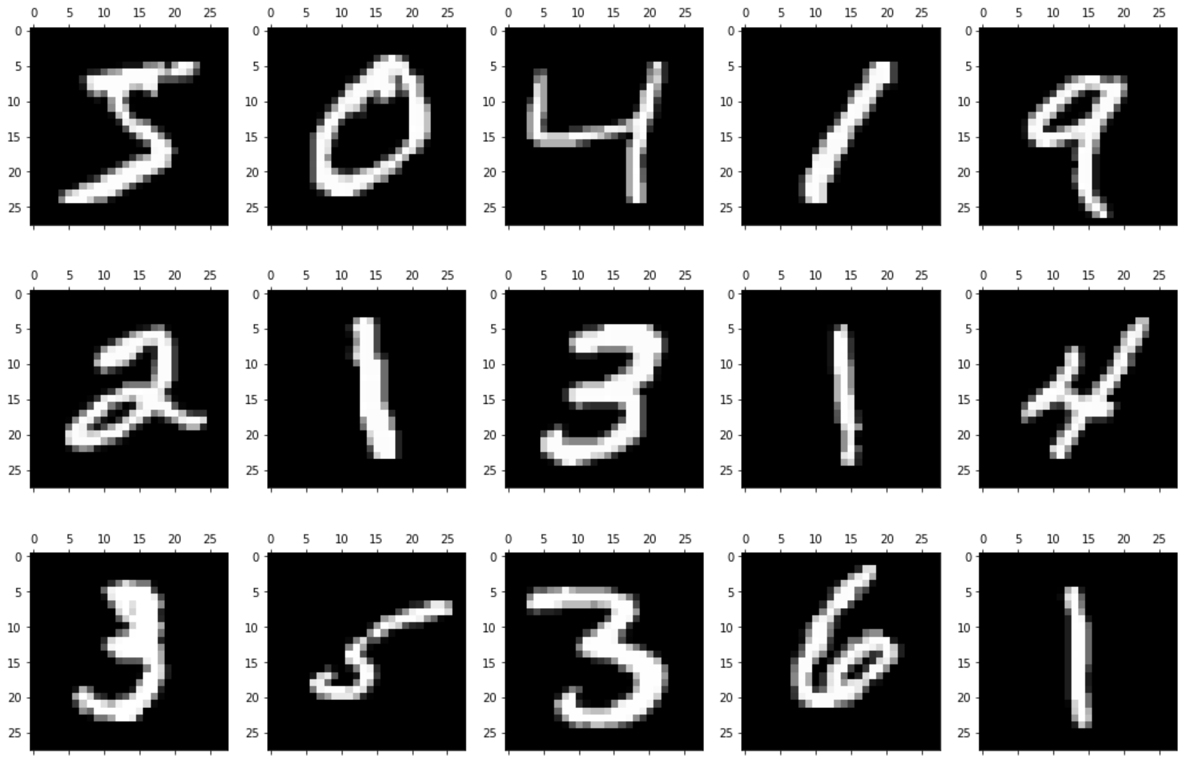
In [38]:

```
figure,axes = plt.subplots(nrows=3, ncols=5)
figure.set_size_inches(18,12)

plt.gray()
print("label={}".format(y_train[0:15]))

col = 0
for row in range(0,3):
    col = row * 5
    axes[row][0].matshow(X_train[col])
    axes[row][1].matshow(X_train[col+1])
    axes[row][2].matshow(X_train[col+2])
    axes[row][3].matshow(X_train[col+3])
    axes[row][4].matshow(X_train[col+4])
```

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



X_train의 데이터 정보를 하나 보기

In [39]:

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

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

Out[39]:

```
(28, 28)
```

신경망에 맞추어 주기 위해 데이터 전처리

- 학습 데이터
- 테스트 데이터

In [40]:

```
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

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

In [41]:

```
import numpy as np
```

In [42]:

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

```
(60000, 784)
```

Out[42]:

```
(0.0, 1.0)
```

In [43]:

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

In [44]:

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

```
(10000, 784)
```

Out[44]:

```
(0.0, 1.0)
```

출력데이터 검증을 위해 10진수의 값을 One-Hot Encoding을 수행

In [45]:

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

변환 전과 후

In [14]:

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

Out[14]:

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

In [15]:

```
Y_train[0:4]
```

Out[15]:

```
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)
```

딥러닝 만들어 보기

In [16]:

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

In [17]:

```
m = Sequential()
```

In [18]:

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

```
WARNING:tensorflow:From C:\Users\peop\Anaconda3\lib\site-packages\tensorflow_core\py
thon\ops\resource_variable_ops.py:1630: calling BaseResourceVariable.__init__ (from
tensorflow.python.ops.resource_variable_ops) with constraint is deprecated and will
be removed in a future version.
Instructions for updating:
If using Keras pass *_constraint arguments to layers.
```

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, validation_data=(X_test, Y_test),
              epochs=30,
              batch_size=200,
              verbose=1)
```

```
WARNING:tensorflow:From C:\Users\peop\Anaconda3\lib\site-packages\keras\backend\tens
orflow_backend.py:422: The name tf.global_variables is deprecated. Please use tf.com
pat.v1.global_variables instead.

Train on 60000 samples, validate on 10000 samples
Epoch 1/30
60000/60000 [==============================] - 6s 99us/step - loss: 0.2796 - accurac
y: 0.9194 - val_loss: 0.1314 - val_accuracy: 0.9603
Epoch 2/30
60000/60000 [==============================] - 6s 94us/step - loss: 0.0979 - accurac
y: 0.9708 - val_loss: 0.0923 - val_accuracy: 0.9712
Epoch 3/30
60000/60000 [==============================] - 6s 104us/step - loss: 0.0628 - accura
cy: 0.9810 - val_loss: 0.0760 - val_accuracy: 0.9751
Epoch 4/30
60000/60000 [==============================] - 6s 96us/step - loss: 0.0437 - accurac
y: 0.9866 - val_loss: 0.0879 - val_accuracy: 0.9739
Epoch 5/30
60000/60000 [==============================] - 6s 93us/step - loss: 0.0319 - accurac
y: 0.9904 - val_loss: 0.0692 - val_accuracy: 0.9799
Epoch 6/30
60000/60000 [==============================] - 5s 91us/step - loss: 0.0224 - accurac
y: 0.9933 - val_loss: 0.0719 - val_accuracy: 0.9804
Epoch 7/30
60000/60000 [==============================] - 6s 93us/step - loss: 0.0157 - accurac
y: 0.9953 - val_loss: 0.0689 - val_accuracy: 0.9803
Epoch 8/30
60000/60000 [==============================] - 6s 98us/step - loss: 0.0152 - accurac
y: 0.9955 - val_loss: 0.0665 - val_accuracy: 0.9812
Epoch 9/30
60000/60000 [==============================] - 6s 97us/step - loss: 0.0115 - accurac
y: 0.9965 - val_loss: 0.0862 - val_accuracy: 0.9762
Epoch 10/30
60000/60000 [==============================] - 6s 95us/step - loss: 0.0095 - accurac
y: 0.9971 - val_loss: 0.0832 - val_accuracy: 0.9788
Epoch 11/30
60000/60000 [==============================] - 6s 92us/step - loss: 0.0099 - accurac
y: 0.9967 - val_loss: 0.1047 - val_accuracy: 0.9752
Epoch 12/30
60000/60000 [==============================] - 6s 95us/step - loss: 0.0106 - accurac
y: 0.9965 - val_loss: 0.0836 - val_accuracy: 0.9804
Epoch 13/30
60000/60000 [==============================] - 6s 92us/step - loss: 0.0074 - accurac
y: 0.9976 - val_loss: 0.0835 - val_accuracy: 0.9813
Epoch 14/30
60000/60000 [==============================] - 6s 93us/step - loss: 0.0044 - accurac
y: 0.9986 - val_loss: 0.0881 - val_accuracy: 0.9815
Epoch 15/30
60000/60000 [==============================] - 5s 90us/step - loss: 0.0042 - accurac
y: 0.9986 - val_loss: 0.0883 - val_accuracy: 0.9805
Epoch 16/30
60000/60000 [==============================] - 5s 90us/step - loss: 0.0121 - accurac
```

```
y: 0.9959 - val_loss: 0.0865 - val_accuracy: 0.9813
Epoch 17/30
60000/60000 [==============================] - 5s 89us/step - loss: 0.0061 - accurac
y: 0.9981 - val_loss: 0.0972 - val_accuracy: 0.9785
Epoch 18/30
60000/60000 [==============================] - 5s 91us/step - loss: 0.0091 - accurac
y: 0.9970 - val_loss: 0.0918 - val_accuracy: 0.9786
Epoch 19/30
60000/60000 [==============================] - 5s 91us/step - loss: 0.0033 - accurac
y: 0.9991 - val_loss: 0.0870 - val_accuracy: 0.9822
Epoch 20/30
60000/60000 [==============================] - 5s 90us/step - loss: 0.0020 - accurac
y: 0.9994 - val_loss: 0.0866 - val_accuracy: 0.9818
Epoch 21/30
60000/60000 [==============================] - 5s 91us/step - loss: 0.0070 - accurac
y: 0.9981 - val_loss: 0.0994 - val_accuracy: 0.9802
Epoch 22/30
60000/60000 [==============================] - 5s 91us/step - loss: 0.0095 - accurac
y: 0.9972 - val_loss: 0.0870 - val_accuracy: 0.9811
Epoch 23/30
60000/60000 [==============================] - 5s 90us/step - loss: 0.0040 - accurac
y: 0.9987 - val_loss: 0.0860 - val_accuracy: 0.9831
Epoch 24/30
60000/60000 [==============================] - 6s 92us/step - loss: 0.0039 - accurac
y: 0.9988 - val_loss: 0.0872 - val_accuracy: 0.9826
Epoch 25/30
60000/60000 [==============================] - 5s 90us/step - loss: 0.0025 - accurac
y: 0.9992 - val_loss: 0.0966 - val_accuracy: 0.9803
Epoch 26/30
60000/60000 [==============================] - 6s 92us/step - loss: 0.0029 - accurac
y: 0.9991 - val_loss: 0.1046 - val_accuracy: 0.9807: 0.0029 - accuracy: 0. - ETA: 0s
- los
Epoch 27/30
60000/60000 [==============================] - 6s 92us/step - loss: 0.0070 - accurac
y: 0.9980 - val_loss: 0.1346 - val_accuracy: 0.9774
Epoch 28/30
60000/60000 [==============================] - 5s 92us/step - loss: 0.0089 - accurac
y: 0.9970 - val_loss: 0.0964 - val_accuracy: 0.9813
Epoch 29/30
60000/60000 [==============================] - 5s 90us/step - loss: 0.0034 - accurac
y: 0.9989 - val_loss: 0.0928 - val_accuracy: 0.9821
Epoch 30/30
60000/60000 [==============================] - 6s 92us/step - loss: 0.0025 - accurac
y: 0.9993 - val_loss: 0.0952 - val_accuracy: 0.9819
```

In [24]:

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

```
10000/10000 [==============================] - 1s 64us/step
Test Accuracy : 0.9819
```

In [25]:

```
pred = m.predict(X_test)
```

In [26]:

```
print( pred.shape )
print( pred[1] )
```

```
(10000, 10)
[1.5944571e-12 2.2818198e-09 1.0000000e+00 2.2257220e-11 2.0903933e-19
 2.5591343e-18 3.7277166e-15 5.9855830e-20 1.9057743e-14 2.6691286e-27]
```

In [ ]:

```

```