6. Workshop 1-2 : MNIST - CNN

> **[Reference] :**
>
> - FRANÇOIS CHOLLET, **Deep Learning with Python**, Chapter 5, Section 1, Manning, 2018. (https://tanthiamhuat.files.wordpress.com/2018/03/deeplearningwithpython.pdf (https://tanthiamhuat.files.wordpress.com/2018/03/deeplearningwithpython.pdf
> - 李飛飛教授：Convolutional Neural Networks (教學投影片) (http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture5.pdf (http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture5.pdf))
> - 李飛飛教授：Convolutional Neural Networks (CNNs / ConvNets) (https://cs231n.github.io/convolutional-networks/ (https://cs231n.github.io/convolutional-networks/))
> - `tf.keras.layers.Conv2D` (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D (https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D))

In [1]:

```python
import keras
keras.__version__
```

```
/Users/macmini1/anaconda3/lib/python3.6/site-packages/h
5py/__init__.py:36: FutureWarning: Conversion of the se
cond argument of issubdtype from `float` to `np.floatin
g` is deprecated. In future, it will be treated as `np.
float64 == np.dtype(float).type`.
  from ._conv import register_converters as _register_c
onverters
Using TensorFlow backend.
```

Out[1]:

```
'2.2.4'
```

Using convnet (Convolutional Neural Network, CNN) to classify MNIST digits :

- **The 6 lines of code below show you what a basic convnet looks like. It's a stack of `Conv2D` and `MaxPooling2D` layers.**
- Importantly, a convnet takes as input tensors of shape `(image_height, image_width, image_channels)` (not including the batch dimension).
- In our case, we will configure our convnet to process inputs of size `(28, 28, 1)`, which is the format of MNIST images. We do this via passing the argument `input_shape=(28, 28, 1)` to our first layer.

In [2]:

```python
from keras import layers
from keras import models

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_sha
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
```

Let's display the architecture of our convnet so far:

In [3]:

```python
model.summary()
```

```
_________________________________________________________________
__________
Layer (type)                 Output Shape
Param #
=================================================================
==========
conv2d_1 (Conv2D)            (None, 28, 28, 32)
320
_________________________________________________________________
__________
max_pooling2d_1 (MaxPooling2 (None, 14, 14, 32)
0
_________________________________________________________________
__________
conv2d_2 (Conv2D)            (None, 12, 12, 64)
18496
_________________________________________________________________
__________
max_pooling2d_2 (MaxPooling2 (None, 6, 6, 64)
0
_________________________________________________________________
__________
conv2d_3 (Conv2D)            (None, 4, 4, 64)
36928
=================================================================
==========
Total params: 55,744
Trainable params: 55,744
Non-trainable params: 0
_________________________________________________________________
__________
```

You can see above that the output of every `Conv2D` and `MaxPooling2D` layer is a 3D tensor of shape `(height, width, channels)`. The width and height dimensions tend to shrink as we go deeper in the network. The number of channels is controlled by the first argument passed to the `Conv2D` layers (e.g. 32 or 64).

The next step would be to feed our last output tensor (of shape `(3, 3, 64)`) into a densely-connected classifier network like those you are already familiar with: a stack of `Dense` layers. These classifiers process vectors, which are 1D, whereas our current output is a 3D tensor. So first, we will have to flatten our 3D outputs to 1D, and then add a few `Dense` layers on top:

In [4]:

```
model.add(layers.Flatten())
model.add(layers.Dense(128, activation='relu'))
model.add(layers.Dense(128, activation='relu'))
model.add(layers.Dense(10, activation='softmax'))
```

We are going to do 10-way classification, so we use a final layer with 10 outputs and a softmax activation. Now here's what our network looks like:

In [5]:

```
1  model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv2d_1 (Conv2D)            (None, 28, 28, 32)        320       
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 14, 14, 32)        0         
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 12, 12, 64)        18496     
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 6, 6, 64)          0         
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 4, 4, 64)          36928     
_________________________________________________________________
flatten_1 (Flatten)          (None, 1024)              0         
_________________________________________________________________
dense_1 (Dense)              (None, 128)               131200    
_________________________________________________________________
dense_2 (Dense)              (None, 128)               16512     
_________________________________________________________________
dense_3 (Dense)              (None, 10)                1290      
=================================================================
Total params: 204,746
Trainable params: 204,746
Non-trainable params: 0
_________________________________________________________________
```

As you can see, our `(3, 3, 64)` outputs were flattened into vectors of shape `(576,)`, before going through two `Dense` layers.

Now, let's train our convnet on the MNIST digits. We will reuse a lot of the code we have already covered in the MNIST example from Chapter 2.

In [6]:

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

(train_images, train_labels), (test_images, test_labels) = mnist

train_images = train_images.reshape((60000, 28, 28, 1))
train_images = train_images.astype('float32') / 255

test_images = test_images.reshape((10000, 28, 28, 1))
test_images = test_images.astype('float32') / 255

train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

In [7]:

```
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(train_images, train_labels, epochs=5, batch_size=256)
```

```
Epoch 1/5
60000/60000 [==============================] - 60s 1ms/
step - loss: 0.3182 - acc: 0.8990
Epoch 2/5
60000/60000 [==============================] - 58s 970u
s/step - loss: 0.0630 - acc: 0.9803
Epoch 3/5
60000/60000 [==============================] - 59s 981u
s/step - loss: 0.0400 - acc: 0.9876
Epoch 4/5
60000/60000 [==============================] - 58s 964u
s/step - loss: 0.0289 - acc: 0.9909
Epoch 5/5
60000/60000 [==============================] - 59s 980u
s/step - loss: 0.0220 - acc: 0.9931
```

Out[7]:

```
<keras.callbacks.History at 0x1202ac550>
```

Let's evaluate the model on the test data:

In [8]:

```
test_loss, test_acc = model.evaluate(test_images, test_labels)
```

```
10000/10000 [==============================] - 4s 363u
s/step
```

In [9]:

```
test_acc
```

Out[9]:

```
0.9926
```

While our densely-connected network from Chapter 2 had a test accuracy of 97.8%, our basic convnet has a test accuracy of 99.3%: we decreased our error rate by 68% (relative). Not bad!

Prediction

In [10]:

```
test_predict = model.predict(test_images)
test_predict
```

Out[10]:

```
array([[4.8063242e-09, 1.6099914e-08, 3.0468024e-07,
..., 9.9999475e-01,
        9.3160729e-08, 8.2932428e-07],
       [3.6947682e-07, 2.5143825e-07, 9.9999940e-01,
..., 2.0474855e-09,
        1.0590493e-09, 1.3229516e-10],
       [5.1350646e-08, 9.9999571e-01, 5.6822856e-08,
..., 3.7499601e-06,
        1.8813036e-08, 4.2574914e-08],
       ...,
       [7.5333576e-13, 6.4846675e-09, 6.9656597e-10,
..., 5.9569025e-08,
        2.7049472e-07, 2.8499809e-07],
       [8.6955410e-08, 2.4490532e-08, 2.1937913e-10,
..., 5.7876974e-08,
        5.5694396e-05, 2.4246023e-07],
       [2.6132991e-06, 3.2795384e-09, 2.9133821e-07,
..., 4.2779116e-13,
        4.0433997e-07, 2.7286309e-09]], dtype=float32)
```

In [11]:

```
import numpy as np
test_predict_result = np.array([np.argmax(test_predict[i]) for i
test_predict_result
```

Out[11]:

```
array([7, 2, 1, ..., 4, 5, 6])
```

In [12]:

```
test_labels_result = np.array([np.argmax(test_labels[i]) for i i
test_labels_result
```

Out[12]:

```
array([7, 2, 1, ..., 4, 5, 6])
```

Confusion Matrix

In [13]:

```
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns; sns.set()

from sklearn.metrics import confusion_matrix

mat = confusion_matrix(test_predict_result, test_labels_result)

plt.figure(figsize=(10,8))
sns.heatmap(mat, square=False, annot=True, cbar=True)
plt.xlabel('predicted value')
plt.ylabel('true value');
```