convnets_mnist_dataset

February 1, 2024

[ ]:
```
import keras
keras.__version__
```

[ ]:
```
'2.15.0'
```

We will use our convnet to classify MNIST digits, a task that you've already been through in Chapter 2, using a densely-connected network (our test accuracy then was 97.8%).

The 6 lines of code below show you what a basic convnet looks like. It's a stack of `Conv2D` and `MaxPooling2D` layers. We'll see in a minute what they do concretely. 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.

[1]:
```
from keras import layers
from keras import models

model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
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:

[2]:
```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 26, 26, 32)        320

 max_pooling2d (MaxPooling2  (None, 13, 13, 32)        0
 D)

 conv2d_1 (Conv2D)           (None, 11, 11, 64)        18496
```

```
 max_pooling2d_1 (MaxPoolin  (None, 5, 5, 64)          0
 g2D)

 conv2d_2 (Conv2D)           (None, 3, 3, 64)          36928

=================================================================
Total params: 55744 (217.75 KB)
Trainable params: 55744 (217.75 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

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:

[3]:
```
model.add(layers.Flatten())
model.add(layers.Dense(64, 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:

[4]:
```
model.summary()
```

```
Model: "sequential"

_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 26, 26, 32)        320

 max_pooling2d (MaxPooling2  (None, 13, 13, 32)        0
 D)

 conv2d_1 (Conv2D)           (None, 11, 11, 64)        18496

 max_pooling2d_1 (MaxPoolin  (None, 5, 5, 64)          0
 g2D)

 conv2d_2 (Conv2D)           (None, 3, 3, 64)          36928

 flatten (Flatten)           (None, 576)               0
```

```
 dense (Dense)               (None, 64)               36928

 dense_1 (Dense)             (None, 10)               650

=================================================================
Total params: 93322 (364.54 KB)
Trainable params: 93322 (364.54 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
```

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.

[5]:
```
from keras.datasets import mnist
from keras.utils import to_categorical
from sklearn.model_selection import train_test_split


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


# Reshape the images for compatibility with Conv2D layer
#train_images = train_images.reshape((train_images.shape[0], 28, 28, 1))


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,num_classes=10)

test_labels = to_categorical(test_labels,num_classes=10)
```

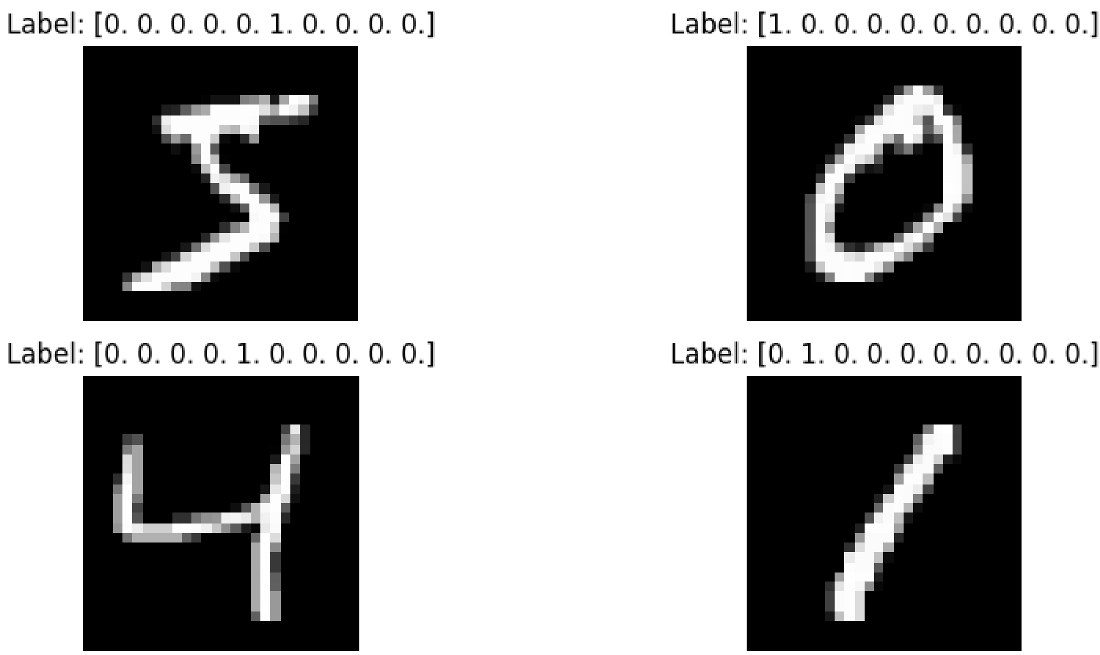
```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-
datasets/mnist.npz
11490434/11490434 [==============================] - 0s 0us/step
```

[6]:
```
print(train_labels.shape)
print(test_labels.shape)
```

```
(60000, 10)
(10000, 10)
```

[7]:
```python
import matplotlib.pyplot as plt
# Display the first four images
plt.figure(figsize=(10, 5))
for i in range(4):
    plt.subplot(2, 2, i + 1)
    plt.imshow(train_images[i], cmap='gray')
    plt.title(f"Label: {train_labels[i]}")
    plt.axis('off')

plt.show()
```



[9]:
```python
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
history = model.fit(train_images, train_labels, epochs=5, batch_size=64) #,␣
↪validation_data=(val_images, val_labels))
```

```
Epoch 1/5
938/938 [==============================] - 49s 51ms/step - loss: 0.1838 -
accuracy: 0.9417
Epoch 2/5
938/938 [==============================] - 48s 51ms/step - loss: 0.0470 -
accuracy: 0.9853
Epoch 3/5
938/938 [==============================] - 49s 52ms/step - loss: 0.0322 -
accuracy: 0.9898
```

```
Epoch 4/5
938/938 [==============================] - 47s 50ms/step - loss: 0.0238 -
accuracy: 0.9927
Epoch 5/5
938/938 [==============================] - 48s 51ms/step - loss: 0.0185 -
accuracy: 0.9941
```

Let's evaluate the model on the test data:

[10]:
```
test_loss, test_acc = model.evaluate(test_images, test_labels)
```

```
313/313 [==============================] - 3s 9ms/step - loss: 0.0268 -
accuracy: 0.9931
```

[11]:
```
test_acc
```

[11]:
```
0.9930999875068665
```