COMP6685_Lab6a_Keras_CNN_MNIST

March 2, 2025

1 Introduction to Convolutional Neural Networks (CNNs)

Convolutional neural networks (CNNs, aka ConvNets) exploit the spatial information in input stimuli (e.g. 2D images), instead of using input as a lineral vector of numbers. The CNN architecture is inspired by the topology and function of the visual cortex, in which a hierarchy of layers process visual information in an incremental way. E.g. our brain starts from the recognition of on/off pixels, then their grouping into lines, their subsequent combination is simple 2D shapes, up to the recognition of compelx 3D shapes and objects.

Specifically, a deep CNN consists of many stacked layers. There are two main types of layers, convolutional and pooling, which typically alternate.

An important principle of the convolution layer is that of receptive fields. That is, a 2D subregion (submatrix) of the input image (matrix) is connected to one hidden unit, and this subregion corresponds to the receptive field of this hidden unit. The next hidden unit will have a receptive field from the adjactent (or partially overlapping) subregion of the input matrix. This corresponds to the convolution mechanism. In Keras, the size of each submatrix is called **kernel size**. This is one of the key hyperparameters in CNNs.

For example, with an MNIST image of 28x28 pixels, and a receptive field (kernel size) of 5x5 pixles feeding into a single hidden unit, and with a shifting of the subregion by 1 pixel (stride length), the next hidden layer will constitute a (feature) map of 23x23 units.

Each input matrix or hidden layer can be connected to multiple **feature maps** in the next hidden layer. All the neurons in the hidden layer of each feature map will share the same weights and biases. This way each feature map layer learns a set of position-independent latent features derived from the image.

The pooling (aka sub-sampling) mechanism is used to group (pool) together the output of a fetaure map. This can be done with Max Pooling (where the highest value of the pooling units is used in the next hidden unit) or Average Pooling (where the average value of the subregion activations is computed).

LeNet for MNIST

One of the pioneering works in Deep Learning and CNN was the original model by Yann LeCun and colleagues (see: Y. LeCun and Y. Bengio, 1995, "Convolutional Networks for Images, Speech, and Time-Series". Brain theory neural networks, vol. 3361). This is referred to as the LeNet. This LeNet was for example used on the MNIST problem, to show the robustness to simple geometric transformations and distortion of the handwritten code. Here we will look at the Keras code for the LeNet on MNIST dataset.

In this exercise we will implemente the code for the specific LeNet network.

2D Convolution module

The definition of a convolution module has this format:

keras.layers.convolutional.Conv2D(filters, kernel_size, padding='valid')

It requires the following parameters: - **filters**, the number of convolution kernels to use (i.e. dimensionality of the output) - **kernel_size**, with two integers specifying the width and height of the 2D convolution window (or a single integer to specify the same value for all spatial dimensions) - **padding**, with **'same'** is used where the area around the input is padded with zeros, resulting in an output with the same size as the input. Instead, **'valid'** is used when the convolution is only computed where the input and the filter fully overlap, resulting in a smaller output size.

2D Pooling module

The definition of a pooling module has this format:

keras.layers.pooling.MaxPooling2D(pool_size=(2, 2), strides=(2, 2))

This uses two main parameters: - **pool_size**, where a tuple of integers represent the verical and horizontal downscaling factors (e.g. (2, 2) will halve the image in each dimension) - **strides**, with the two integers for the stride/dimensions used for the max/average pooling processing

Initialisation for the program

The program starts with the importing of typical Keras and other Python service modules.

[1]:
```
# importing of modules for LeNet CNN
from tensorflow.keras import backend as K
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import MaxPooling2D
from tensorflow.keras.layers import Activation
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Dense
from tensorflow.keras.datasets import mnist
from tensorflow.keras import utils
from tensorflow.keras.optimizers import SGD, RMSprop, Adam

# importing of service libraries
import numpy as np
import matplotlib.pyplot as plt

print('Libraries imported.')
```

```
2025-01-31 15:11:41.538616: I tensorflow/core/platform/cpu_feature_guard.cc:210]
This TensorFlow binary is optimized to use available CPU instructions in
performance-critical operations.
To enable the following instructions: AVX2 AVX512F AVX512_VNNI FMA, in other
operations, rebuild TensorFlow with the appropriate compiler flags.
```

```
Libraries imported.
```

The following variable definitions are needed for the network and training parameters, and the image size.

[ ]:
```
# seed for reproducibility
np.random.seed(1671)

# network and training parameters
N_EPOCH = 20 # later use 20 for better results
BATCH_SIZE = 128
VERBOSE = 1
OPTIMIZER = Adam()
VALIDATION_SPLIT=0.2

IMG_ROWS, IMG_COLS = 28, 28 # input  dimensions of each MNIST image
N_CLASSES = 10  # number of outputs = number of digits
INPUT_SHAPE = (1, IMG_ROWS, IMG_COLS)

print('Main variables initialised.')
```

```
Main variables initialised.
```

LeNet class definition for the CNN model

This code defines the class LeNet for the building of the CNN model.

It has a first convolution module followed by a pooling layer. This convolution layer (with ReLu activation function) has 20 filters, each with a kernel size of 5x5. The output dimension is the same as the input imge of 28x28 units (because we use the **same** padding parameter. The pooling layers uses a regioin of 2x2, with max pooling values.

Another comvolution layer of 50 filters is then pooled.

This layer is then flattened into a one-dimensional layer, followed by a dense layer of 500 units.

Finally, a softmax dense layer in output classifies the images into the 10 number categories.

[3]:
```
#define the convnet
class LeNet:
        @staticmethod
        def build(input_shape, classes):
                model = Sequential()

                # CONV => RELU => POOL
                model.add(Conv2D(20, kernel_size=5, padding="same",␣
 ↪input_shape=input_shape))
                model.add(Activation("relu"))
                model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2),␣
 ↪padding='same'))
```

```
                # CONV => RELU => POOL
                model.add(Conv2D(50, kernel_size=5, padding="same"))
                model.add(Activation("relu"))
                model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2),␣
 ↪padding='same'))

                # Flatten => RELU layers
                model.add(Flatten())
                model.add(Dense(500))
                model.add(Activation("relu"))

                # a softmax classifier
                model.add(Dense(classes))
                model.add(Activation("softmax"))

                return model

print('LeNet class defined.')
```

```
LeNet class defined.
```

MNIST data loading and processing

This code loads the MNIST dataset, as in the previous labs.

[ ]:
```
# data: shuffled and split between train and test sets
(input_X_train, output_y_train), (input_X_test, output_y_test) = mnist.
 ↪load_data()
# K.set_image_dim_ordering("th")

# consider them as float and normalize
input_X_train = input_X_train.astype('float32')
input_X_test = input_X_test.astype('float32')
input_X_train /= 255
input_X_test /= 255

# we need a 60K x [1 x 28 x 28] shape as input to the CONVNET
input_X_train = input_X_train[:, np.newaxis, :, :]
input_X_test = input_X_test[:, np.newaxis, :, :]

print(input_X_train.shape[0], 'train samples')
print(input_X_test.shape[0], 'test samples')

# convert class vectors to binary class matrices
output_y_train = utils.to_categorical(output_y_train, N_CLASSES)
output_y_test = utils.to_categorical(output_y_test, N_CLASSES)
```

```
60000 train samples
```

```
10000 test samples
```

Model initialisation and compilation

This initialises the model using the LeNet function, and then compiles the network and shows its summary.

[6]:
```
# initialize the optimizer and compile the model
model = LeNet.build(input_shape=INPUT_SHAPE, classes=N_CLASSES)

model.compile(loss="categorical_crossentropy", optimizer=OPTIMIZER,␣
 ↪metrics=["accuracy"])

model.summary()
```

```
Model: "sequential_1"
```

Layer (type)	Output Shape	Param #
conv2d_2 (Conv2D)	(None, 1, 28, 20)	14,020
activation_4 (Activation)	(None, 1, 28, 20)	0
max_pooling2d_2 (MaxPooling2D)	(None, 1, 14, 20)	0
conv2d_3 (Conv2D)	(None, 1, 14, 50)	25,050
activation_5 (Activation)	(None, 1, 14, 50)	0
max_pooling2d_3 (MaxPooling2D)	(None, 1, 7, 50)	0
flatten_1 (Flatten)	(None, 350)	0
dense_2 (Dense)	(None, 500)	175,500
activation_6 (Activation)	(None, 500)	0
dense_3 (Dense)	(None, 10)	5,010
activation_7 (Activation)	(None, 10)	0

```
Total params: 219,580 (857.73 KB)

Trainable params: 219,580 (857.73 KB)
```

```
 Non-trainable params: 0 (0.00 B)
```

Training of the CNN

This line of code trains the model, saving the metrics data in the history variable.

[7]:
```
# training/fitting of the LeNet model


history = model.fit(input_X_train, output_y_train, batch_size=BATCH_SIZE,␣
 ↪epochs=N_EPOCH, verbose=VERBOSE, validation_split=VALIDATION_SPLIT)
```

```
Epoch 1/20
375/375             10s 22ms/step -
accuracy: 0.7928 - loss: 0.7167 - val_accuracy: 0.9628 - val_loss: 0.1265
Epoch 2/20
375/375             11s 29ms/step -
accuracy: 0.9641 - loss: 0.1184 - val_accuracy: 0.9693 - val_loss: 0.0986
Epoch 3/20
375/375             13s 33ms/step -
accuracy: 0.9754 - loss: 0.0829 - val_accuracy: 0.9767 - val_loss: 0.0765
Epoch 4/20
375/375             14s 37ms/step -
accuracy: 0.9789 - loss: 0.0661 - val_accuracy: 0.9769 - val_loss: 0.0799
Epoch 5/20
375/375             16s 42ms/step -
accuracy: 0.9844 - loss: 0.0520 - val_accuracy: 0.9814 - val_loss: 0.0637
Epoch 6/20
375/375             15s 40ms/step -
accuracy: 0.9874 - loss: 0.0408 - val_accuracy: 0.9814 - val_loss: 0.0632
Epoch 7/20
375/375             16s 42ms/step -
accuracy: 0.9885 - loss: 0.0356 - val_accuracy: 0.9808 - val_loss: 0.0643
Epoch 8/20
375/375             13s 36ms/step -
accuracy: 0.9905 - loss: 0.0291 - val_accuracy: 0.9842 - val_loss: 0.0533
Epoch 9/20
375/375             13s 34ms/step -
accuracy: 0.9917 - loss: 0.0248 - val_accuracy: 0.9843 - val_loss: 0.0519
Epoch 10/20
375/375             16s 42ms/step -
accuracy: 0.9935 - loss: 0.0200 - val_accuracy: 0.9829 - val_loss: 0.0611
Epoch 11/20
375/375             15s 41ms/step -
accuracy: 0.9942 - loss: 0.0186 - val_accuracy: 0.9789 - val_loss: 0.0731
Epoch 12/20
375/375             20s 54ms/step -
accuracy: 0.9941 - loss: 0.0180 - val_accuracy: 0.9852 - val_loss: 0.0601
```

```
Epoch 13/20
375/375             17s 46ms/step -
accuracy: 0.9960 - loss: 0.0132 - val_accuracy: 0.9838 - val_loss: 0.0603
Epoch 14/20
375/375             18s 39ms/step -
accuracy: 0.9964 - loss: 0.0115 - val_accuracy: 0.9847 - val_loss: 0.0635
Epoch 15/20
375/375             12s 32ms/step -
accuracy: 0.9967 - loss: 0.0109 - val_accuracy: 0.9861 - val_loss: 0.0562
Epoch 16/20
375/375             14s 37ms/step -
accuracy: 0.9970 - loss: 0.0089 - val_accuracy: 0.9868 - val_loss: 0.0556
Epoch 17/20
375/375             15s 41ms/step -
accuracy: 0.9978 - loss: 0.0070 - val_accuracy: 0.9856 - val_loss: 0.0640
Epoch 18/20
375/375             16s 42ms/step -
accuracy: 0.9975 - loss: 0.0078 - val_accuracy: 0.9843 - val_loss: 0.0662
Epoch 19/20
375/375             15s 40ms/step -
accuracy: 0.9981 - loss: 0.0059 - val_accuracy: 0.9849 - val_loss: 0.0612
Epoch 20/20
375/375             16s 42ms/step -
accuracy: 0.9970 - loss: 0.0079 - val_accuracy: 0.9857 - val_loss: 0.0643
```

Analysis of the training results

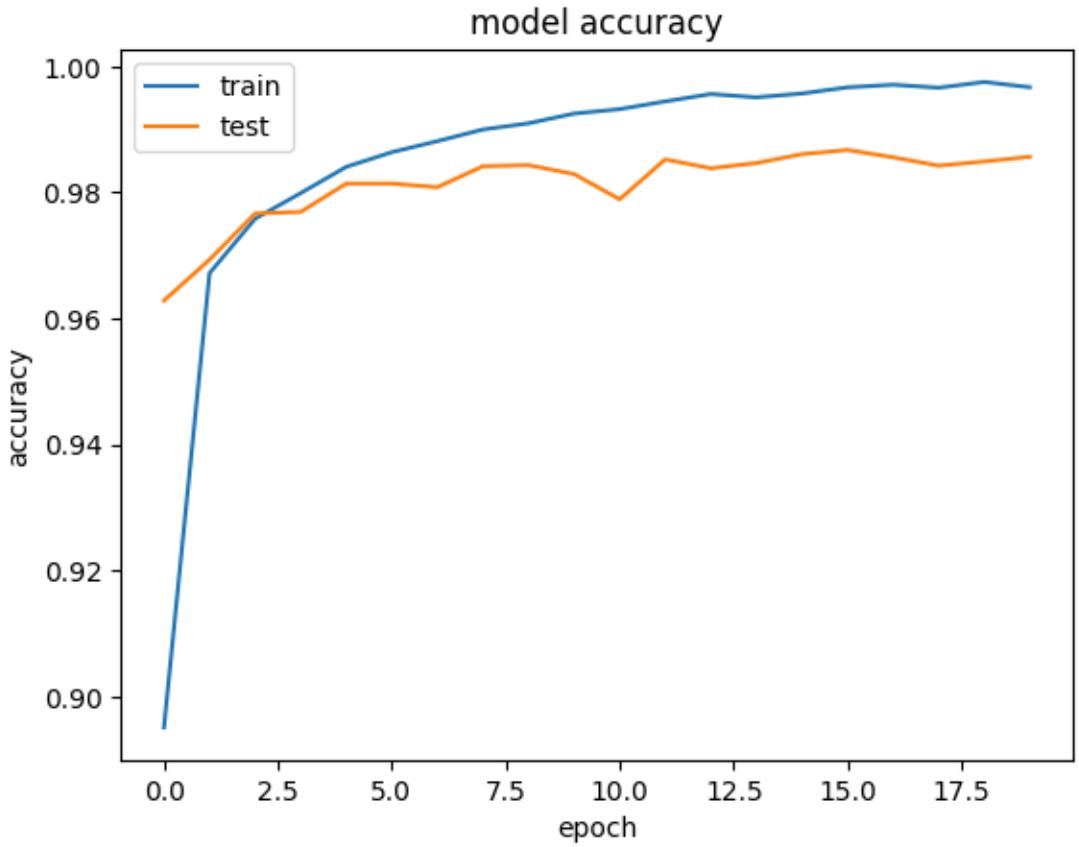
This generates the test scores by evaluating the trained network with the test dataset.

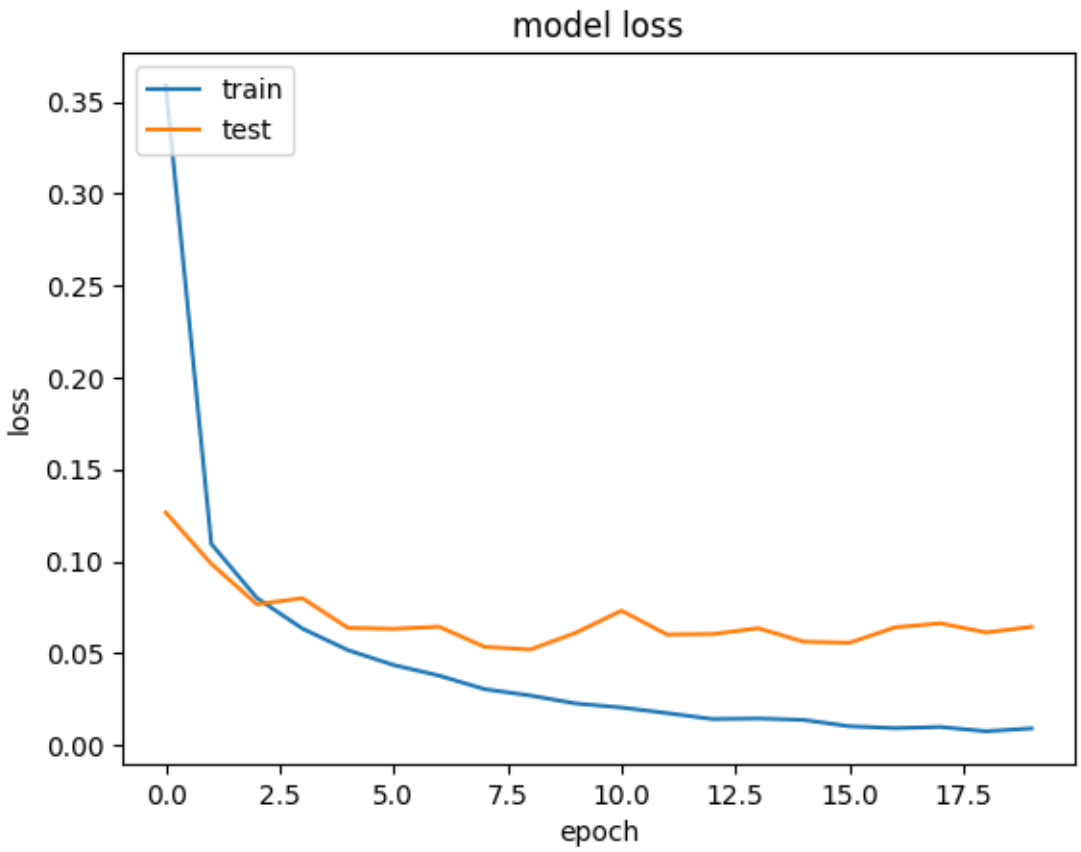
It also plots the accuracy and loss values along the training timescale.

[8]:
```
score = model.evaluate(input_X_test, output_y_test, verbose=VERBOSE)
print("\nTest score/loss:", score[0])
print('Test accuracy:', score[1])

# list all data in history
print(history.history.keys())
# summarize history for accuracy
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
```

```
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

```
313/313             3s 10ms/step -
accuracy: 0.9845 - loss: 0.0635

Test score/loss: 0.05727870389819145
Test accuracy: 0.9868000149726868
dict_keys(['accuracy', 'loss', 'val_accuracy', 'val_loss'])
```





1.1 Saving and Loading a Network Model

It is possible to save the configuration of a specific network model in a JSON file format. This will also allow a later, faster reloading of the model and its training, as in the example below.

[18]:
```
from tensorflow.keras.models import model_from_json


#Creation of of the model configuration in json format
model_json = model.to_json()

#save model in json format into a file
open('CNN_MNIST_architecture.json', 'w').write(model_json)

print('Model definition jsom file saved.')

#model reconstruction from json format
model = model_from_json(model_json)
OPTIMIZER = Adam()
# Now let's compile, summarise and then train/fit the model.
```

```
model.compile(loss="categorical_crossentropy", optimizer=OPTIMIZER,␣
 ↪metrics=["accuracy"], run_eagerly=True)

model.summary()

# let's reduce the number of epoch to 3, for a faster test of the checkpoint␣
 ↪utility
N_EPOCH = 3

history = model.fit(input_X_train, output_y_train, batch_size=BATCH_SIZE,␣
 ↪epochs=N_EPOCH, verbose=VERBOSE, validation_split=VALIDATION_SPLIT)
```

```
Model definition jsom file saved.
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
conv2d_2 (Conv2D)	(None, 1, 28, 20)	14,020
activation_4 (Activation)	(None, 1, 28, 20)	0
max_pooling2d_2 (MaxPooling2D)	(None, 1, 14, 20)	0
conv2d_3 (Conv2D)	(None, 1, 14, 50)	25,050
activation_5 (Activation)	(None, 1, 14, 50)	0
max_pooling2d_3 (MaxPooling2D)	(None, 1, 7, 50)	0
flatten_1 (Flatten)	(None, 350)	0
dense_2 (Dense)	(None, 500)	175,500
activation_6 (Activation)	(None, 500)	0
dense_3 (Dense)	(None, 10)	5,010
activation_7 (Activation)	(None, 10)	0

Total params: 219,580 (857.73 KB)

Trainable params: 219,580 (857.73 KB)

```
 Non-trainable params: 0 (0.00 B)


Epoch 1/3
375/375             49s 130ms/step -
accuracy: 0.7938 - loss: 0.7419 - val_accuracy: 0.9507 - val_loss: 0.1653
Epoch 2/3
375/375             48s 127ms/step -
accuracy: 0.9615 - loss: 0.1255 - val_accuracy: 0.9754 - val_loss: 0.0838
Epoch 3/3
375/375             45s 120ms/step -
accuracy: 0.9755 - loss: 0.0807 - val_accuracy: 0.9776 - val_loss: 0.0762
```

1.2 Checkpoints

Another important utility in Keras is the saving of the trained weights of the network at the end of the training, or at specific timesteps called checkpoints. This will permit, for example, (i) the continuation of the training at a later stage starting from the saved checkpoint weights, (ii) the later inspection of the network performance at the chosen checkpoints, (iii) the saving of the weights of the best version during the training, e.g. for a specified accuracy value.

Saving weights at the end of the training

The first example below shows how to save the weighst at the end of the training.

[22]:
```
from tensorflow.keras.models import load_model

#This saves the weights at the last epoch of the previous training session
model.save('my_model.keras')
print ('Weights saved for final epoch ', N_EPOCH)
print ('Check that there is a weight file in your folder')

#This loads the weight file 'my_model.LeNet'
model = load_model('my_model.keras')
print ('Loading of the saved weights and test using these trained weights')

# let's check that the weighst loaded in this examples have kept the trained␣
  ↪state of the network.
score = model.evaluate(input_X_test, output_y_test, verbose=VERBOSE)
print("\nTest score/loss:", score[0])
print('Test accuracy:', score[1])
```

```
Weights saved for final epoch  3
Check that there is a weight file in your folder
Loading of the saved weights and test using these trained weights
313/313             11s 35ms/step -
accuracy: 0.9781 - loss: 0.0680

Test score/loss: 0.0638914555311203
Test accuracy: 0.9807000160217285
```

Regular checkpoint saving

This example shows how to save the weights at regular inverval checkpoints. The default value for the **ModelCheckpoint** functiion is every 1 epoch.

[25]:
```
# additional checkpoint load code and temp directory definition
from tensorflow.keras.callbacks import ModelCheckpoint
from tensorflow.keras.models import load_model
import os
MODEL_DIR = "./tmp"

# let's reduce the number of epoch to 3, for a faster test of the checkpoint␣
 ↪utility
N_EPOCH = 3


# definition of checkpoint parameters file
if not os.path.exists(MODEL_DIR):
    os.makedirs(MODEL_DIR)
checkpoint = ModelCheckpoint(filepath=os.path.join(MODEL_DIR, "my_model-{epoch:
 ↪02d}.keras"))

print ('Training with checkpoint at each epoch. Check that at the end of each␣
 ↪epoch, there is a weight file in the tmp folder')
model.fit(input_X_train, output_y_train, batch_size=BATCH_SIZE, epochs=N_EPOCH,␣
 ↪validation_split=0.5, callbacks=[checkpoint])

score = model.evaluate(input_X_test, output_y_test, verbose=VERBOSE)
print("\nTest score/loss:", score[0])
print('Test accuracy:', score[1])

# Let's load the weights of the 2nd epoch, and check the scores
model = load_model(os.path.join(MODEL_DIR,'my_model-02.keras'))
print ('Weights loaded for epoch 2 .')
print ('New test using these trained weights.')

# let's check that the weights loaded in this examples have kept the trained␣
 ↪state of the network.
score = model.evaluate(input_X_test, output_y_test, verbose=VERBOSE)
print("\nTest score/loss:", score[0])
print('Test accuracy:', score[1])
```

```
Training with checkpoint at each epoch. Check that at the end of each epoch,
there is a weight file in the tmp folder
Epoch 1/3
235/235             33s 140ms/step -
accuracy: 0.9905 - loss: 0.0313 - val_accuracy: 0.9818 - val_loss: 0.0600
Epoch 2/3
```

```
235/235             30s 130ms/step -
accuracy: 0.9925 - loss: 0.0251 - val_accuracy: 0.9785 - val_loss: 0.0701
Epoch 3/3
235/235             32s 136ms/step -
accuracy: 0.9912 - loss: 0.0266 - val_accuracy: 0.9827 - val_loss: 0.0602
313/313             9s 28ms/step -
accuracy: 0.9835 - loss: 0.0548

Test score/loss: 0.05068347230553627
Test accuracy: 0.9848999977111816
Weights loaded for epoch 2 .
New test using these trained weights.
313/313             9s 28ms/step -
accuracy: 0.9795 - loss: 0.0667

Test score/loss: 0.058457206934690475
Test accuracy: 0.9812999963760376
```

1.3 Using TensorBoard to Visualise Model Performance

Keras provides a callback for saving the training and test metrics, as well as the activation histograms for the different layers in the model.

The saved data can then be visualized with the TensorBoad launched at the command line:

[26]:
```
from tensorflow.keras.callbacks import TensorBoard

N_EPOCH = 20

#keras.callbacks.TensorBoard(log_dir='./logs', histogram_freq=0,␣
 ↪write_graph=True, write_images=False)
tensorboard = TensorBoard(log_dir='./logs', histogram_freq=0, write_graph=True,␣
 ↪write_images=False)

model.fit(input_X_train, output_y_train, batch_size=BATCH_SIZE, epochs=N_EPOCH,␣
 ↪validation_split=0.5, callbacks=[tensorboard])

# Launch of TensorBoard to visualise the performance of the mdoel using the␣
 ↪saved data


# Launch tensorboard in colab:
%load_ext tensorboard
%tensorboard --logdir ./logs

# or launch tensorboard if you're running the exercise in your computer
#!tensorboard "--logdir=./logs" --host localhost --port 6006
# Launch Chrome browser and go to localhost:6006 to view tensorboard
```

```
Epoch 17/20
235/235             32s 135ms/step -
accuracy: 0.9994 - loss: 0.0018 - val_accuracy: 0.9823 - val_loss: 0.0870
Epoch 18/20
235/235             37s 156ms/step -
accuracy: 0.9978 - loss: 0.0067 - val_accuracy: 0.9813 - val_loss: 0.0869
Epoch 19/20
235/235             36s 154ms/step -
accuracy: 0.9964 - loss: 0.0102 - val_accuracy: 0.9833 - val_loss: 0.0830
Epoch 20/20
235/235             36s 155ms/step -
accuracy: 0.9992 - loss: 0.0034 - val_accuracy: 0.9835 - val_loss: 0.0825
```

```
<IPython.core.display.HTML object>
```

1.4 Conclusions

We have learned to train our first, real **deep** neural network, using the well known LeCun CNN for the MNIST dataset. Later we will explore the use of deeper networks and more complex dataset.