Lab Report TP-IMA4103
TP 1: MNIST Digit Recognition

Yakine KLABI

Outline :

Introduction :

In this lab, we focus on the MNIST dataset, which contains images of handwritten digits. Our goal is to write Python scripts that will be able to recognize the digits from an input image. We will use two implementations, namely a CNN architecture (in the second application), and compare their performances.

I) Application I :

Please note: for your convenience, the outputs of this part have been grouped by exercise, not in one single part of the report.

The objective of this part is to write a Python script that is able to recognize the digit from an input image. The digits range from 0 to 9: as a result, there are 10 classes to predict. To train our model, we are going to use the MNIST dataset, which contains 60 000 images in the training set and 10 000 images in the validation set.

a) Code for this part

The following code (Figure 1) is the one we will analyze in this first part of our report. To improve our understanding of the code, we decided to rename a few variables (please see table below) :

Name in the original code	Name in our code
`X_train`	`train_images`
`Y_train`	`train_labels`
`X_test`	`test_images`
`Y_test`	`test_labels`

We also added minor modifications to the code (namely to implement loops: for example, we added parameters in the function "`trainAndPredictMLP(train_images, train_labels, test_images, test_labels, neurons, batch_size)`", such as "`neurons`", or "`batch_size`". It allowed us to perform several trainings of the model by using a loop, to solve exercises 1 and 2. The code here is "frozen" in the state it has after solving exercise 4: if you want to check the solving of a particular exercise, please uncomment the corresponding parts.

```
import keras
import tensorflow as tf
import time
from keras.datasets import mnist
#from keras.utils import np_utils
from keras.utils import to_categorical
from keras import layers

################################################################################
def baseline_model(num_pixels, num_classes, neurons):

    #TODO - Application 1 - Step 6a - Initialize the sequential model
    model = tf.keras.models.Sequential()    # Modify this
```

```
    #TODO - Application 1 - Step 6b - Define a hidden dense layer with 8 neurons
    model.add(layers.Dense(neurons, input_dim=num_pixels, kernel_initializer='normal',
activation='relu')) # the number of neurons has to be modified here
    # for exercise 1

    #TODO - Application 1 - Step 6c - Define the output dense layer
    model.add(layers.Dense(num_classes, kernel_initializer='normal',
activation='softmax'))

    # TODO - Application 1 - Step 6d - Compile the model
    model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])

    # the following line has been used to solve exercise 3:
    # model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['mse'])

    return model
################################################################################

def trainAndPredictMLP(train_images, train_labels, test_images, test_labels, neurons,
batch_size):

    #TODO - Application 1 - Step 3 - Reshape the MNIST dataset - Transform the images to
1D vectors of floats (28x28 pixels  to  784 elements)
    num_pixels = train_images.shape[1] * train_images.shape[2]
    train_images = train_images.reshape((train_images.shape[0],
num_pixels)).astype('float32')
    test_images = test_images.reshape((test_images.shape[0],
num_pixels)).astype('float32')

    #TODO - Application 1 - Step 4 - Normalize the input values
    train_images = train_images / 255
    test_images = test_images / 255

    #TODO - Application 1 - Step 5 - Transform the classes labels into a binary matrix
    train_labels = to_categorical(train_labels)
    test_labels = to_categorical(test_labels)
    num_classes = test_labels.shape[1]

    #TODO - Application 1 - Step 6 - Build the model architecture - Call the
baseline_model function

    model = baseline_model(num_pixels, num_classes, neurons)  #Modify this

    #TODO - Application 1 - Step 7 - Train the model
    start_time=time.time()
    model.fit(train_images, train_labels, validation_data=(test_images, test_labels),
epochs=10, batch_size=batch_size, verbose=2)
    end_time=time.time()

    # the following line has been used to solve exercise 4
```

```
    model.save_weights('./application1.h5')

    convergence_time=end_time-start_time
    #TODO - Application 1 - Step 8 - System evaluation - compute and display the
prediction error
    scores = model.evaluate(test_images, test_labels, verbose=2)
    print("Baseline Error: {:.2f}".format(100-scores[1]*100))
   #This line has also been used to solve exercise 3 instead of the line above using
the baseline error
    #print(f"mse of batch size {batch_size}is: {scores[1]}")
    print(f"the convergence time is : {convergence_time:.2f} seconds\n")



    return
################################################################################

def main():

    #TODO - Application 1 - Step 1 - Load the MNIST dataset in Keras
    (train_images, train_labels), (test_images, test_labels) = mnist.load_data()

    #TODO - Application 1 - Step 2 - Train and predict on a MLP - Call the
trainAndPredictMLP function
    # loop used to solve exercise 1
    # neurons =[8,16,32,64,128]
    # for neuron in neurons :
    #     print("system performance with :", neuron,"neurons")
    #     mlp = trainAndPredictMLP(train_images, train_labels, test_images, test_labels,
neuron, batch_size=200)

    ### MLP : application 1, exercise 4
    neurons=8
    batch_size=200
    mlp = trainAndPredictMLP(train_images, train_labels, test_images, test_labels,
neurons, batch_size)

    # loop used to solve exercise 2
    # batch_sizes =[32,64,128,256,512]
    # for batch_size in batch_sizes :
    #     print("system performance with the following batch size :", batch_size)
    #     mlp = trainAndPredictMLP(train_images, train_labels, test_images, test_labels,
neurons, batch_size)

    #TODO - Application 2 - Step 1 - Train and predict on a CNN - Call the
trainAndPredictCNN function
    cnn = trainAndPredictCNN(train_images, train_labels, test_images, test_labels)

    return
################################################################################if
__name__ == '__main__':
    main()
```

b) *Answer for all the exercises related to the script, and outputs of the script related to each exercise*

Exercise 1: Modify the number of neurons on the hidden layer as specified in Table 1. What can be observed regarding the system accuracy? How about the convergence time necessary for the system to train a model?

In the first exercise, we are asked to train the model several times, but with a different number of neurons in the hidden layer each time. To do so, we implemented the yellow part of the code above: we added the parameter "`neurons`" to the `trainAndPredictMLP` function. In the main function, we defined an array "`neurons`" that contains all the different values of the number of neurons on the hidden layer that we want to use to train our model. We then used a "`for`" loop to perform the training. To obtain the convergence time (time necessary to perform the training), we recorded the time when the training started thanks to the function "`time.time()`", and the time when it ended. We then subtracted those two parameters and we obtained the following output and result (figure 2):

```
300/300 - 2s - loss: 1.1997 - accuracy: 0.6730 - val_loss: 0.52
90 - val_accuracy: 0.8612 - 2s/epoch - 6ms/step
Epoch 2/10
300/300 - 1s - loss: 0.4468 - accuracy: 0.8813 - val_loss: 0.36
50 - val_accuracy: 0.9003 - 922ms/epoch - 3ms/step
Epoch 3/10
300/300 - 1s - loss: 0.3557 - accuracy: 0.9027 - val_loss: 0.32
42 - val_accuracy: 0.9095 - 949ms/epoch - 3ms/step
Epoch 4/10
300/300 - 1s - loss: 0.3253 - accuracy: 0.9103 - val_loss: 0.30
54 - val_accuracy: 0.9140 - 621ms/epoch - 2ms/step
Epoch 5/10
300/300 - 0s - loss: 0.3074 - accuracy: 0.9147 - val_loss: 0.29
63 - val_accuracy: 0.9174 - 486ms/epoch - 2ms/step
Epoch 6/10
300/300 - 1s - loss: 0.2953 - accuracy: 0.9179 - val_loss: 0.28
35 - val_accuracy: 0.9207 - 613ms/epoch - 2ms/step
Epoch 7/10
300/300 - 1s - loss: 0.2864 - accuracy: 0.9205 - val_loss: 0.27
71 - val_accuracy: 0.9236 - 752ms/epoch - 3ms/step
Epoch 8/10
300/300 - 1s - loss: 0.2788 - accuracy: 0.9225 - val_loss: 0.27
29 - val_accuracy: 0.9236 - 628ms/epoch - 2ms/step
Epoch 9/10
300/300 - 1s - loss: 0.2726 - accuracy: 0.9239 - val_loss: 0.27
00 - val_accuracy: 0.9252 - 550ms/epoch - 2ms/step
Epoch 10/10
300/300 - 1s - loss: 0.2674 - accuracy: 0.9256 - val_loss: 0.26
40 - val_accuracy: 0.9261 - 651ms/epoch - 2ms/step
313/313 - 1s - loss: 0.2640 - accuracy: 0.9261 - 525ms/epoch -
2ms/step
Baseline Error: 7.39
the convergence time is : 8.15 seconds
```

```
system performance with : 16 neurons
Epoch 1/10
300/300 - 1s - loss: 0.9446 - accuracy: 0.7575 - val_loss: 0.40
31 - val_accuracy: 0.8946 - 1s/epoch - 4ms/step
Epoch 2/10
300/300 - 1s - loss: 0.3570 - accuracy: 0.9014 - val_loss: 0.30
16 - val_accuracy: 0.9173 - 662ms/epoch - 2ms/step
Epoch 3/10
300/300 - 1s - loss: 0.2977 - accuracy: 0.9159 - val_loss: 0.26
83 - val_accuracy: 0.9232 - 678ms/epoch - 2ms/step
Epoch 4/10
300/300 - 1s - loss: 0.2685 - accuracy: 0.9238 - val_loss: 0.24
75 - val_accuracy: 0.9304 - 698ms/epoch - 2ms/step
Epoch 5/10
300/300 - 1s - loss: 0.2493 - accuracy: 0.9296 - val_loss: 0.23
82 - val_accuracy: 0.9318 - 594ms/epoch - 2ms/step
Epoch 6/10
300/300 - 1s - loss: 0.2355 - accuracy: 0.9339 - val_loss: 0.22
63 - val_accuracy: 0.9349 - 584ms/epoch - 2ms/step
Epoch 7/10
300/300 - 1s - loss: 0.2239 - accuracy: 0.9370 - val_loss: 0.21
80 - val_accuracy: 0.9375 - 666ms/epoch - 2ms/step
Epoch 8/10
300/300 - 0s - loss: 0.2153 - accuracy: 0.9392 - val_loss: 0.20
90 - val_accuracy: 0.9394 - 478ms/epoch - 2ms/step
Epoch 9/10
300/300 - 1s - loss: 0.2075 - accuracy: 0.9413 - val_loss: 0.20
55 - val_accuracy: 0.9408 - 652ms/epoch - 2ms/step
Epoch 10/10
300/300 - 1s - loss: 0.2010 - accuracy: 0.9431 - val_loss: 0.20
18 - val_accuracy: 0.9417 - 809ms/epoch - 3ms/step
313/313 - 1s - loss: 0.2018 - accuracy: 0.9417 - 640ms/epoch -
2ms/step
Baseline Error: 5.83
the convergence time is : 7.27 seconds
```

```
system performance with : 32 neurons
Epoch 1/10
300/300 - 1s - loss: 0.7385 - accuracy: 0.8179 - val_loss: 0.31
84 - val_accuracy: 0.9119 - 1s/epoch - 5ms/step
Epoch 2/10
300/300 - 1s - loss: 0.2906 - accuracy: 0.9181 - val_loss: 0.25
10 - val_accuracy: 0.9276 - 766ms/epoch - 3ms/step
Epoch 3/10
300/300 - 1s - loss: 0.2395 - accuracy: 0.9321 - val_loss: 0.21
71 - val_accuracy: 0.9369 - 864ms/epoch - 3ms/step
Epoch 4/10
300/300 - 1s - loss: 0.2089 - accuracy: 0.9402 - val_loss: 0.19
70 - val_accuracy: 0.9431 - 722ms/epoch - 2ms/step
Epoch 5/10
300/300 - 1s - loss: 0.1873 - accuracy: 0.9461 - val_loss: 0.18
19 - val_accuracy: 0.9455 - 1s/epoch - 4ms/step
Epoch 6/10
300/300 - 1s - loss: 0.1698 - accuracy: 0.9514 - val_loss: 0.16
84 - val_accuracy: 0.9509 - 1s/epoch - 3ms/step
Epoch 7/10
300/300 - 1s - loss: 0.1569 - accuracy: 0.9554 - val_loss: 0.15
83 - val_accuracy: 0.9532 - 557ms/epoch - 2ms/step
Epoch 8/10
300/300 - 1s - loss: 0.1465 - accuracy: 0.9575 - val_loss: 0.15
49 - val_accuracy: 0.9521 - 881ms/epoch - 3ms/step
Epoch 9/10
300/300 - 1s - loss: 0.1374 - accuracy: 0.9604 - val_loss: 0.14
48 - val_accuracy: 0.9561 - 606ms/epoch - 2ms/step
Epoch 10/10
300/300 - 1s - loss: 0.1294 - accuracy: 0.9622 - val_loss: 0.14
12 - val_accuracy: 0.9576 - 570ms/epoch - 2ms/step
313/313 - 1s - loss: 0.1412 - accuracy: 0.9576 - 569ms/epoch -
2ms/step
Baseline Error: 4.24
the convergence time is : 8.59 seconds
```

```
system performance with : 64 neurons
Epoch 1/10
300/300 - 1s - loss: 0.5940 - accuracy: 0.8554 - val_loss: 0.28
73 - val_accuracy: 0.9205 - 1s/epoch - 5ms/step
Epoch 2/10
300/300 - 1s - loss: 0.2610 - accuracy: 0.9260 - val_loss: 0.22
31 - val_accuracy: 0.9371 - 909ms/epoch - 3ms/step
Epoch 3/10
300/300 - 1s - loss: 0.2070 - accuracy: 0.9418 - val_loss: 0.18
78 - val_accuracy: 0.9450 - 1s/epoch - 4ms/step
Epoch 4/10
300/300 - 1s - loss: 0.1717 - accuracy: 0.9510 - val_loss: 0.16
22 - val_accuracy: 0.9518 - 1s/epoch - 4ms/step
Epoch 5/10
300/300 - 1s - loss: 0.1474 - accuracy: 0.9576 - val_loss: 0.14
43 - val_accuracy: 0.9564 - 810ms/epoch - 3ms/step
Epoch 6/10
300/300 - 1s - loss: 0.1275 - accuracy: 0.9637 - val_loss: 0.13
38 - val_accuracy: 0.9626 - 1s/epoch - 3ms/step
Epoch 7/10
300/300 - 1s - loss: 0.1133 - accuracy: 0.9673 - val_loss: 0.11
96 - val_accuracy: 0.9634 - 985ms/epoch - 3ms/step
Epoch 8/10
300/300 - 1s - loss: 0.1009 - accuracy: 0.9708 - val_loss: 0.11
31 - val_accuracy: 0.9669 - 1s/epoch - 4ms/step
Epoch 9/10
300/300 - 1s - loss: 0.0912 - accuracy: 0.9736 - val_loss: 0.10
75 - val_accuracy: 0.9671 - 750ms/epoch - 2ms/step
Epoch 10/10
300/300 - 1s - loss: 0.0826 - accuracy: 0.9761 - val_loss: 0.10
20 - val_accuracy: 0.9689 - 685ms/epoch - 2ms/step
313/313 - 1s - loss: 0.1020 - accuracy: 0.9689 - 543ms/epoch -
2ms/step
Baseline Error: 3.11
the convergence time is : 10.01 seconds
```

```
system performance with : 128 neurons
Epoch 1/10
300/300 - 2s - loss: 0.4863 - accuracy: 0.8773 - val_loss: 0.25
Epoch 2/10
300/300 - 1s - loss: 0.2193 - accuracy: 0.9381 - val_loss: 0.18
87 - val_accuracy: 0.9454 - 1s/epoch - 5ms/step
Epoch 3/10
300/300 - 2s - loss: 0.1646 - accuracy: 0.9531 - val_loss: 0.14
41 - val_accuracy: 0.9583 - 2s/epoch - 6ms/step
Epoch 4/10
300/300 - 1s - loss: 0.1312 - accuracy: 0.9628 - val_loss: 0.13
15 - val_accuracy: 0.9614 - 1s/epoch - 4ms/step
Epoch 5/10
300/300 - 1s - loss: 0.1063 - accuracy: 0.9697 - val_loss: 0.10
75 - val_accuracy: 0.9675 - 1s/epoch - 4ms/step
Epoch 6/10
300/300 - 1s - loss: 0.0887 - accuracy: 0.9746 - val_loss: 0.09
81 - val_accuracy: 0.9707 - 1s/epoch - 4ms/step
Epoch 7/10
300/300 - 1s - loss: 0.0758 - accuracy: 0.9786 - val_loss: 0.09
30 - val_accuracy: 0.9723 - 1s/epoch - 4ms/step
Epoch 8/10
300/300 - 1s - loss: 0.0654 - accuracy: 0.9811 - val_loss: 0.08
85 - val_accuracy: 0.9735 - 1s/epoch - 4ms/step
Epoch 9/10
300/300 - 1s - loss: 0.0570 - accuracy: 0.9840 - val_loss: 0.08
49 - val_accuracy: 0.9750 - 1s/epoch - 4ms/step
Epoch 10/10
300/300 - 1s - loss: 0.0500 - accuracy: 0.9859 - val_loss: 0.08
12 - val_accuracy: 0.9750 - 1s/epoch - 4ms/step
313/313 - 1s - loss: 0.0812 - accuracy: 0.9750 - 530ms/epoch -
2ms/step
Baseline Error: 2.50
the convergence time is : 13.44 seconds
```

Number of neurons	8	16	32	64	128

System Accuracy	0.9261	0.9417	0.9576	0.9689	0.9750
Convergence time (s)	8.15	7.27	8.59	10.01	13.44

Figure 2 : Table of results for exercise 1

The first thing we can notice is that the accuracy seems to increase with the number of neurons contained by the hidden layer. It seems quite logical to us: the more neurons there are, the more complex patterns the neural network can learn/the better it learns. As for the convergence time, it seems to increase with the number of neurons. However, it does not increase "linearly" mathematically speaking.
In fact, this increase seems obvious because adding more neurons to the hidden layers of the network generates more operations and more calculations, which requires more computations. This is why the convergence time increases with the number of neurons.

Exercise 2: Modify the batch size used to train the model as specified in Table 2. Select a value of 8 neurons for the hidden layer. What can be observed regarding the system accuracy?

Let's take a look at the orange part of the code above. The exercise asks us to train the model using different batch sizes. To do so, we have used the same strategy as in exercise 1, which is to create a loop (see below) that does all the training processes we need in one run of the program.

```
    batch_sizes =[32,64,128,256,512]
    for batch_size in batch_sizes :
       print("system performance with the following batch size :", batch_size)
       mlp = trainAndPredictMLP(train_images, train_labels, test_images, test_labels,
neurons, batch_size)
```

Extract from the code above: loop to train the model with different batch sizes

The outputs of our code are shown below. They are summarized in the following table (figure 3):

```
system performance with the following batch size : 32
Epoch 1/10
1875/1875 - 14s - loss: 0.5999 - accuracy: 0.8301 - val_loss: 0.3415 - val_accuracy: 0.9015 - 14s/epoch - 8ms/step
Epoch 2/10
1875/1875 - 11s - loss: 0.3265 - accuracy: 0.9071 - val_loss: 0.3014 - val_accuracy: 0.9142 - 11s/epoch - 6ms/step
Epoch 3/10
1875/1875 - 8s - loss: 0.3004 - accuracy: 0.9147 - val_loss: 0.2896 - val_accuracy: 0.9191 - 8s/epoch - 4ms/step
Epoch 4/10
1875/1875 - 13s - loss: 0.2869 - accuracy: 0.9195 - val_loss: 0.2847 - val_accuracy: 0.9196 - 13s/epoch - 7ms/step
Epoch 5/10
1875/1875 - 13s - loss: 0.2782 - accuracy: 0.9219 - val_loss: 0.2876 - val_accuracy: 0.9172 - 13s/epoch - 7ms/step
Epoch 6/10
1875/1875 - 11s - loss: 0.2718 - accuracy: 0.9232 - val_loss: 0.2799 - val_accuracy: 0.9190 - 11s/epoch - 6ms/step
Epoch 7/10
1875/1875 - 12s - loss: 0.2657 - accuracy: 0.9253 - val_loss: 0.2730 - val_accuracy: 0.9244 - 12s/epoch - 6ms/step
Epoch 8/10
1875/1875 - 12s - loss: 0.2610 - accuracy: 0.9265 - val_loss: 0.2701 - val_accuracy: 0.9254 - 12s/epoch - 7ms/step
Epoch 9/10
1875/1875 - 13s - loss: 0.2565 - accuracy: 0.9280 - val_loss: 0.2709 - val_accuracy: 0.9258 - 13s/epoch - 7ms/step
Epoch 10/10
1875/1875 - 13s - loss: 0.2532 - accuracy: 0.9287 - val_loss: 0.2733 - val_accuracy: 0.9225 - 13s/epoch - 7ms/step
313/313 - 2s - loss: 0.2733 - accuracy: 0.9225 - 2s/epoch - 6ms/step
Baseline Error: 7.75
the convergence time is 119.87452912330627 seconds
```

```
system performance with the following batch size : 64
Epoch 1/10
938/938 - 7s - loss: 0.7368 - accuracy: 0.8009 - val_loss: 0.3736 - val_accuracy: 0.8947 - 7s/epoch - 7ms/step
Epoch 2/10
938/938 - 6s - loss: 0.3502 - accuracy: 0.9003 - val_loss: 0.3144 - val_accuracy: 0.9112 - 6s/epoch - 7ms/step
Epoch 3/10
938/938 - 6s - loss: 0.3109 - accuracy: 0.9112 - val_loss: 0.2968 - val_accuracy: 0.9168 - 6s/epoch - 7ms/step
Epoch 4/10
938/938 - 6s - loss: 0.2936 - accuracy: 0.9164 - val_loss: 0.2866 - val_accuracy: 0.9197 - 6s/epoch - 7ms/step
Epoch 5/10
938/938 - 6s - loss: 0.2831 - accuracy: 0.9195 - val_loss: 0.2795 - val_accuracy: 0.9222 - 6s/epoch - 7ms/step
Epoch 6/10
938/938 - 7s - loss: 0.2755 - accuracy: 0.9222 - val_loss: 0.2755 - val_accuracy: 0.9241 - 7s/epoch - 7ms/step
Epoch 7/10
938/938 - 7s - loss: 0.2693 - accuracy: 0.9242 - val_loss: 0.2708 - val_accuracy: 0.9229 - 7s/epoch - 8ms/step
Epoch 8/10
938/938 - 7s - loss: 0.2646 - accuracy: 0.9251 - val_loss: 0.2707 - val_accuracy: 0.9251 - 7s/epoch - 8ms/step
Epoch 9/10
938/938 - 7s - loss: 0.2611 - accuracy: 0.9266 - val_loss: 0.2674 - val_accuracy: 0.9242 - 7s/epoch - 8ms/step
Epoch 10/10
938/938 - 7s - loss: 0.2569 - accuracy: 0.9273 - val_loss: 0.2623 - val_accuracy: 0.9262 - 7s/epoch - 7ms/step
313/313 - 1s - loss: 0.2623 - accuracy: 0.9262 - 903ms/epoch - 3ms/step
Baseline Error: 7.38
the convergence time is 67.12950158119202 seconds
system performance with the following batch size : 128
Epoch 1/10
469/469 - 3s - loss: 0.9532 - accuracy: 0.7358 - val_loss: 0.4515 - val_accuracy: 0.8720 - 3s/epoch - 7ms/step
Epoch 2/10
469/469 - 3s - loss: 0.4156 - accuracy: 0.8787 - val_loss: 0.3641 - val_accuracy: 0.8929 - 3s/epoch - 7ms/step
Epoch 3/10
469/469 - 3s - loss: 0.3622 - accuracy: 0.8944 - val_loss: 0.3377 - val_accuracy: 0.9009 - 3s/epoch - 7ms/step
Epoch 4/10
469/469 - 2s - loss: 0.3368 - accuracy: 0.9020 - val_loss: 0.3262 - val_accuracy: 0.9069 - 2s/epoch - 5ms/step
Epoch 5/10
469/469 - 2s - loss: 0.3198 - accuracy: 0.9073 - val_loss: 0.3145 - val_accuracy: 0.9086 - 2s/epoch - 5ms/step
Epoch 6/10
469/469 - 2s - loss: 0.3079 - accuracy: 0.9113 - val_loss: 0.3055 - val_accuracy: 0.9150 - 2s/epoch - 5ms/step
Epoch 7/10
469/469 - 2s - loss: 0.2973 - accuracy: 0.9149 - val_loss: 0.2972 - val_accuracy: 0.9161 - 2s/epoch - 5ms/step
Epoch 8/10
469/469 - 2s - loss: 0.2891 - accuracy: 0.9176 - val_loss: 0.2918 - val_accuracy: 0.9192 - 2s/epoch - 5ms/step
Epoch 9/10
469/469 - 2s - loss: 0.2834 - accuracy: 0.9199 - val_loss: 0.2872 - val_accuracy: 0.9213 - 2s/epoch - 5ms/step
Epoch 10/10
469/469 - 2s - loss: 0.2781 - accuracy: 0.9216 - val_loss: 0.2849 - val_accuracy: 0.9212 - 2s/epoch - 4ms/step
313/313 - 1s - loss: 0.2849 - accuracy: 0.9212 - 900ms/epoch - 3ms/step
Baseline Error: 7.88
the convergence time is 25.393945455551147 seconds
```

```
system performance with the following batch size : 256
Epoch 1/10
235/235 - 2s - loss: 1.3310 - accuracy: 0.6032 - val_loss: 0.6583 - val_accuracy: 0.8206 - 2s/epoch - 8ms/step
Epoch 2/10
235/235 - 1s - loss: 0.5269 - accuracy: 0.8572 - val_loss: 0.4214 - val_accuracy: 0.8829 - 1s/epoch - 5ms/step
Epoch 3/10
235/235 - 2s - loss: 0.3952 - accuracy: 0.8905 - val_loss: 0.3533 - val_accuracy: 0.9023 - 2s/epoch - 8ms/step
Epoch 4/10
235/235 - 2s - loss: 0.3489 - accuracy: 0.9023 - val_loss: 0.3228 - val_accuracy: 0.9075 - 2s/epoch - 9ms/step
Epoch 5/10
235/235 - 2s - loss: 0.3241 - accuracy: 0.9076 - val_loss: 0.3101 - val_accuracy: 0.9115 - 2s/epoch - 9ms/step
Epoch 6/10
235/235 - 2s - loss: 0.3088 - accuracy: 0.9120 - val_loss: 0.2960 - val_accuracy: 0.9153 - 2s/epoch - 8ms/step
Epoch 7/10
235/235 - 2s - loss: 0.2980 - accuracy: 0.9154 - val_loss: 0.2916 - val_accuracy: 0.9168 - 2s/epoch - 8ms/step
Epoch 8/10
235/235 - 1s - loss: 0.2904 - accuracy: 0.9179 - val_loss: 0.2834 - val_accuracy: 0.9194 - 1s/epoch - 5ms/step
Epoch 9/10
235/235 - 2s - loss: 0.2836 - accuracy: 0.9199 - val_loss: 0.2794 - val_accuracy: 0.9206 - 2s/epoch - 6ms/step
Epoch 10/10
235/235 - 2s - loss: 0.2784 - accuracy: 0.9215 - val_loss: 0.2733 - val_accuracy: 0.9231 - 2s/epoch - 8ms/step
313/313 - 1s - loss: 0.2733 - accuracy: 0.9231 - 756ms/epoch - 2ms/step
Baseline Error: 7.69
the convergence time is 17.468879222869873 seconds
system performance with the following batch size : 512
Epoch 1/10
118/118 - 2s - loss: 1.7648 - accuracy: 0.4765 - val_loss: 1.0723 - val_accuracy: 0.7776 - 2s/epoch - 14ms/step
Epoch 2/10
118/118 - 1s - loss: 0.7582 - accuracy: 0.8181 - val_loss: 0.5422 - val_accuracy: 0.8634 - 778ms/epoch - 7ms/step
Epoch 3/10
118/118 - 1s - loss: 0.4864 - accuracy: 0.8743 - val_loss: 0.4128 - val_accuracy: 0.8891 - 809ms/epoch - 7ms/step
Epoch 4/10
118/118 - 1s - loss: 0.4017 - accuracy: 0.8911 - val_loss: 0.3622 - val_accuracy: 0.8996 - 758ms/epoch - 6ms/step
Epoch 5/10
118/118 - 1s - loss: 0.3623 - accuracy: 0.9000 - val_loss: 0.3368 - val_accuracy: 0.9044 - 864ms/epoch - 7ms/step
Epoch 6/10
118/118 - 1s - loss: 0.3400 - accuracy: 0.9057 - val_loss: 0.3222 - val_accuracy: 0.9067 - 1s/epoch - 11ms/step
Epoch 7/10
118/118 - 1s - loss: 0.3251 - accuracy: 0.9085 - val_loss: 0.3097 - val_accuracy: 0.9112 - 1s/epoch - 11ms/step
Epoch 8/10
118/118 - 1s - loss: 0.3139 - accuracy: 0.9112 - val_loss: 0.3024 - val_accuracy: 0.9138 - 1s/epoch - 10ms/step
Epoch 9/10
118/118 - 1s - loss: 0.3058 - accuracy: 0.9136 - val_loss: 0.2955 - val_accuracy: 0.9155 - 914ms/epoch - 8ms/step
Epoch 10/10
118/118 - 1s - loss: 0.2986 - accuracy: 0.9162 - val_loss: 0.2900 - val_accuracy: 0.9161 - 1s/epoch - 10ms/step
313/313 - 1s - loss: 0.2900 - accuracy: 0.9161 - 1s/epoch - 4ms/step
Baseline Error: 8.39
the convergence time is 10.958871364593506 seconds
```

Batch size	32	64	128	256	512
System Accuracy	0.9225	0.9262	0.9212	0.9231	0.9161

Figure 3 : Table of results for exercise 2

In our case, we did not really notice any relationship between the batch_size and the accuracy of our model. However, we think that as the batch_size represents the number of samples from the dataset that is used to train the model in one forward/backward pass, when the batch_size is relatively important, the training time decreases. However, it has several side effects: indeed, the higher the batch size, the more memory space we might need to perform the training. Moreover, a large batch_size can cause a decrease in accuracy (the neural network does not manage to generalize very well the patterns it finds). On the other hand, having a relatively small batch_size can be useful to converge more quickly towards good results. However, the training can take more memory space with

this setting, and can cause overfitting because of a possibly noisy weight update. Thus, it is important to find a good tradeoff between accuracy, speed and memory space taken to reach the best performance possible according to the hardware's capacity.

Exercise 3: Modify the metric used to determine the system performance from accuracy to mean square error (mse). Has this parameter any influence over the training process?
The MSE is defined as follows:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - y_{i_{predicted}})^2$$

Here, the subtraction represents the difference between the ground truth score and the predicted score. To implement the mse as a metrics in the code, we have written the line in pink, where we replace "`accuracy`" by "`mse`" in the `"metrics"` parameter of the compile function. We have used the same process as in the previous exercise, where we have trained our model using different batch sizes every time.

The following outputs are summarized in the table below:

```
1875/1875 - 3s - loss: 0.5881 - mse: 0.0258 - val_loss: 0.3447 - val_mse: 0.0152 - 3s/epoch - 2ms/step
Epoch 2/10
1875/1875 - 2s - loss: 0.3266 - mse: 0.0143 - val_loss: 0.3024 - val_mse: 0.0132 - 2s/epoch - 1ms/step
Epoch 3/10
1875/1875 - 3s - loss: 0.3040 - mse: 0.0133 - val_loss: 0.3023 - val_mse: 0.0134 - 3s/epoch - 2ms/step
Epoch 4/10
1875/1875 - 3s - loss: 0.2923 - mse: 0.0127 - val_loss: 0.2897 - val_mse: 0.0125 - 3s/epoch - 2ms/step
Epoch 5/10
1875/1875 - 3s - loss: 0.2841 - mse: 0.0123 - val_loss: 0.2840 - val_mse: 0.0124 - 3s/epoch - 2ms/step
Epoch 6/10
1875/1875 - 3s - loss: 0.2775 - mse: 0.0121 - val_loss: 0.2814 - val_mse: 0.0122 - 3s/epoch - 2ms/step
Epoch 7/10
1875/1875 - 3s - loss: 0.2722 - mse: 0.0119 - val_loss: 0.2806 - val_mse: 0.0124 - 3s/epoch - 2ms/step
Epoch 8/10
1875/1875 - 3s - loss: 0.2695 - mse: 0.0117 - val_loss: 0.2795 - val_mse: 0.0121 - 3s/epoch - 2ms/step
Epoch 9/10
1875/1875 - 3s - loss: 0.2651 - mse: 0.0116 - val_loss: 0.2773 - val_mse: 0.0120 - 3s/epoch - 2ms/step
Epoch 10/10
1875/1875 - 3s - loss: 0.2632 - mse: 0.0115 - val_loss: 0.2807 - val_mse: 0.0122 - 3s/epoch - 2ms/step
313/313 - 0s - loss: 0.2807 - mse: 0.0122 - 388ms/epoch - 1ms/step
mse of batch size 32is: 0.012228482402861118
the convergence time is : 29.40 seconds
```



```
system performance with the following batch size : 64
Epoch 1/10
938/938 - 2s - loss: 0.7241 - mse: 0.0318 - val_loss: 0.3736 - val_mse: 0.0167 - 2s/epoch - 2ms/step
Epoch 2/10
938/938 - 2s - loss: 0.3514 - mse: 0.0155 - val_loss: 0.3162 - val_mse: 0.0138 - 2s/epoch - 2ms/step
Epoch 3/10
938/938 - 1s - loss: 0.3150 - mse: 0.0138 - val_loss: 0.2988 - val_mse: 0.0129 - 1s/epoch - 1ms/step
Epoch 4/10
938/938 - 2s - loss: 0.2966 - mse: 0.0130 - val_loss: 0.2935 - val_mse: 0.0128 - 2s/epoch - 2ms/step
Epoch 5/10
938/938 - 2s - loss: 0.2849 - mse: 0.0124 - val_loss: 0.2854 - val_mse: 0.0123 - 2s/epoch - 2ms/step
Epoch 6/10
938/938 - 2s - loss: 0.2762 - mse: 0.0120 - val_loss: 0.2782 - val_mse: 0.0119 - 2s/epoch - 2ms/step
Epoch 7/10
938/938 - 2s - loss: 0.2695 - mse: 0.0117 - val_loss: 0.2734 - val_mse: 0.0116 - 2s/epoch - 2ms/step
Epoch 8/10
938/938 - 2s - loss: 0.2641 - mse: 0.0115 - val_loss: 0.2701 - val_mse: 0.0116 - 2s/epoch - 2ms/step
Epoch 9/10
938/938 - 2s - loss: 0.2593 - mse: 0.0113 - val_loss: 0.2719 - val_mse: 0.0117 - 2s/epoch - 2ms/step
Epoch 10/10
938/938 - 2s - loss: 0.2551 - mse: 0.0111 - val_loss: 0.2730 - val_mse: 0.0118 - 2s/epoch - 2ms/step
313/313 - 0s - loss: 0.2730 - mse: 0.0118 - 389ms/epoch - 1ms/step
mse of batch size 64is: 0.011778264306485653
the convergence time is : 15.96 seconds
```

```
system performance with the following batch size : 128
Epoch 1/10
469/469 - 1s - loss: 0.9530 - mse: 0.0413 - val_loss: 0.4280 - val_mse: 0.0190 - 1s/epoch - 3ms/step
Epoch 2/10
469/469 - 1s - loss: 0.3866 - mse: 0.0170 - val_loss: 0.3438 - val_mse: 0.0151 - 852ms/epoch - 2ms/step
Epoch 3/10
469/469 - 1s - loss: 0.3373 - mse: 0.0147 - val_loss: 0.3185 - val_mse: 0.0138 - 861ms/epoch - 2ms/step
Epoch 4/10
469/469 - 1s - loss: 0.3159 - mse: 0.0137 - val_loss: 0.3023 - val_mse: 0.0131 - 868ms/epoch - 2ms/step
Epoch 5/10
469/469 - 1s - loss: 0.3038 - mse: 0.0132 - val_loss: 0.2913 - val_mse: 0.0127 - 918ms/epoch - 2ms/step
Epoch 6/10
469/469 - 1s - loss: 0.2941 - mse: 0.0128 - val_loss: 0.2854 - val_mse: 0.0124 - 854ms/epoch - 2ms/step
Epoch 7/10
469/469 - 1s - loss: 0.2862 - mse: 0.0124 - val_loss: 0.2793 - val_mse: 0.0120 - 884ms/epoch - 2ms/step
Epoch 8/10
469/469 - 1s - loss: 0.2796 - mse: 0.0121 - val_loss: 0.2759 - val_mse: 0.0120 - 847ms/epoch - 2ms/step
Epoch 9/10
469/469 - 1s - loss: 0.2736 - mse: 0.0119 - val_loss: 0.2754 - val_mse: 0.0120 - 862ms/epoch - 2ms/step
Epoch 10/10
469/469 - 1s - loss: 0.2688 - mse: 0.0117 - val_loss: 0.2666 - val_mse: 0.0115 - 862ms/epoch - 2ms/step
313/313 - 0s - loss: 0.2666 - mse: 0.0115 - 384ms/epoch - 1ms/step
mse of batch size 128is: 0.011471958830952644
the convergence time is : 9.13 seconds
```

```
system performance with the following batch size : 256
Epoch 1/10
235/235 - 1s - loss: 1.3177 - mse: 0.0571 - val_loss: 0.6546 - val_mse: 0.0293 - 865ms/epoch - 4ms/step
Epoch 2/10
235/235 - 1s - loss: 0.5453 - mse: 0.0245 - val_loss: 0.4455 - val_mse: 0.0197 - 505ms/epoch - 2ms/step
Epoch 3/10
235/235 - 1s - loss: 0.4139 - mse: 0.0184 - val_loss: 0.3701 - val_mse: 0.0162 - 516ms/epoch - 2ms/step
Epoch 4/10
235/235 - 0s - loss: 0.3548 - mse: 0.0157 - val_loss: 0.3293 - val_mse: 0.0145 - 478ms/epoch - 2ms/step
Epoch 5/10
235/235 - 1s - loss: 0.3234 - mse: 0.0143 - val_loss: 0.3105 - val_mse: 0.0136 - 501ms/epoch - 2ms/step
Epoch 6/10
235/235 - 1s - loss: 0.3058 - mse: 0.0134 - val_loss: 0.2996 - val_mse: 0.0131 - 510ms/epoch - 2ms/step
Epoch 7/10
235/235 - 1s - loss: 0.2948 - mse: 0.0129 - val_loss: 0.2910 - val_mse: 0.0127 - 504ms/epoch - 2ms/step
Epoch 8/10
235/235 - 1s - loss: 0.2864 - mse: 0.0125 - val_loss: 0.2852 - val_mse: 0.0124 - 503ms/epoch - 2ms/step
Epoch 9/10
235/235 - 1s - loss: 0.2799 - mse: 0.0122 - val_loss: 0.2810 - val_mse: 0.0122 - 505ms/epoch - 2ms/step
Epoch 10/10
235/235 - 0s - loss: 0.2747 - mse: 0.0120 - val_loss: 0.2796 - val_mse: 0.0121 - 496ms/epoch - 2ms/step
313/313 - 0s - loss: 0.2796 - mse: 0.0121 - 394ms/epoch - 1ms/step
mse of batch size 256is: 0.012083479203283787
the convergence time is : 5.50 seconds
```

```
system performance with the following batch size : 512
Epoch 1/10
118/118 - 1s - loss: 1.7463 - mse: 0.0737 - val_loss: 1.0546 - val_mse: 0.0484 - 854ms/epoch - 7ms/step
Epoch 2/10
118/118 - 1s - loss: 0.7773 - mse: 0.0354 - val_loss: 0.5846 - val_mse: 0.0264 - 503ms/epoch - 4ms/step
Epoch 3/10
118/118 - 1s - loss: 0.5309 - mse: 0.0239 - val_loss: 0.4639 - val_mse: 0.0208 - 624ms/epoch - 5ms/step
Epoch 4/10
118/118 - 1s - loss: 0.4472 - mse: 0.0199 - val_loss: 0.4014 - val_mse: 0.0177 - 611ms/epoch - 5ms/step
Epoch 5/10
118/118 - 1s - loss: 0.3864 - mse: 0.0169 - val_loss: 0.3580 - val_mse: 0.0156 - 632ms/epoch - 5ms/step
Epoch 6/10
118/118 - 1s - loss: 0.3556 - mse: 0.0155 - val_loss: 0.3359 - val_mse: 0.0146 - 637ms/epoch - 5ms/step
Epoch 7/10
118/118 - 0s - loss: 0.3380 - mse: 0.0147 - val_loss: 0.3236 - val_mse: 0.0139 - 287ms/epoch - 2ms/step
Epoch 8/10
118/118 - 0s - loss: 0.3249 - mse: 0.0141 - val_loss: 0.3139 - val_mse: 0.0136 - 250ms/epoch - 2ms/step
Epoch 9/10
118/118 - 0s - loss: 0.3151 - mse: 0.0137 - val_loss: 0.3058 - val_mse: 0.0132 - 228ms/epoch - 2ms/step
Epoch 10/10
118/118 - 0s - loss: 0.3070 - mse: 0.0134 - val_loss: 0.3000 - val_mse: 0.0130 - 218ms/epoch - 2ms/step
313/313 - 0s - loss: 0.3000 - mse: 0.0130 - 339ms/epoch - 1ms/step
mse of batch size 512is: 0.012964930385351181
the convergence time is : 4.96 seconds
```

Batch size	32	64	128	256	512
Last Mean Square Error recorded in the training process	0.0114	0.0117	0.0120	0.0121	0.0127

Figure 4 : Table of results for exercise 3

What we notice is that the MSE is very low, meaning that our model seems to perform well on the data. When we look at a metrics, we have to be particularly careful when it comes to its meaning: we are looking for MSEs close to zero, but to accuracies close to 1! However, we notice that the baseline error seems very high. It might be because it is computed according to an "accuracy" metric, whereas we use the MSE metric there, which quantifies the errors of our model… and not its good guesses (which correspond to the "baseline error" metrics there!).

Exercise 4: Using the default parameters specified at Application 1 (8 neurons in hidden layer, 10 epochs and a batch size of 200 images), train the model and save the associated weights in a file (model.save_weights), so it can be load anytime to perform further tests.

Let's take a look at the lines in blue in the code above. The exercise asks us to save the weights of the model using the function `save_weights`. It is what we implemented, by saving our model's weights in the file "`application1.h5`". We have used the HDF5 format, widely used in machine learning to store large amounts of data. In the next exercise, we will be able to reload the weights saved in this part to perform a test of our model!

Exercise 5: Write a novel Python script that loads the saved weights (model.load_weights) at Exercise 4 and make a prediction on the first 5 images of the testing dataset (mnist.test_images()).

This exercise asks us to reload the weights, computed after training our model, and to perform a prediction on the five first images of the MNIST test dataset. The code we have implemented is presented below:

```
# prediction on the first 5 images of the testing dataset (mnist.test_images())

import keras
import tensorflow as tf
import time
from keras.datasets import mnist
#from keras.utils import np_utils
from keras.utils import to_categorical
from keras import layers
import numpy as np
########################################################################
def baseline_model(num_pixels, num_classes, neurons):

    #TODO - Application 1 - Step 6a - Initialize the sequential model
    model = tf.keras.models.Sequential()    # Modify this
```

```
    #TODO - Application 1 - Step 6b - Define a hidden dense layer with 8 neurons
    model.add(layers.Dense(neurons, input_dim=num_pixels, kernel_initializer='normal',
activation='relu')) # the number of neurons has to be modified here
    # for exercise 1

    #TODO - Application 1 - Step 6c - Define the output dense layer
    model.add(layers.Dense(num_classes, kernel_initializer='normal',
activation='softmax'))

    # TODO - Application 1 - Step 6d - Compile the model
    model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])

    # the following line has been used to solve exercise 3:
    # model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['mse'])

    return model
########################################################################

def PredictMLP(train_images, train_labels, test_images, test_labels, neurons,
batch_size):

    #TODO - Application 1 - Step 3 - Reshape the MNIST dataset - Transform the images to
1D vectors of floats (28x28 pixels  to  784 elements)
    num_pixels = train_images.shape[1] * train_images.shape[2]
    train_images = train_images.reshape((train_images.shape[0],
num_pixels)).astype('float32')
    test_images = test_images.reshape((test_images.shape[0],
num_pixels)).astype('float32')

    #TODO - Application 1 - Step 4 - Normalize the input values
    train_images = train_images / 255
    test_images = test_images / 255

    #TODO - Application 1 - Step 5 - Transform the classes labels into a binary matrix
    train_labels = to_categorical(train_labels)
    test_labels = to_categorical(test_labels)
    num_classes = test_labels.shape[1]

    print("size of the initial test dataset (images) :" + str(len(test_images)))
    print("size of the initial test dataset (labels) :" + str(len(test_labels)))

    test_images_cropped = test_images[0:5]
    test_labels_cropped = test_labels[0:5]

    print("size of the test dataset containing the 5 first images (images) :" +
str(len(test_images_cropped)))
    print("size of the test dataset containing the 5 first images :" +
str(len(test_labels_cropped)))

    #TODO - Application 1 - Step 6 - Build the model architecture - Call the
baseline_model function
```

```
    model = baseline_model(num_pixels, num_classes, neurons)  #Modify this
    model.load_weights('./application1.h5')

    model.summary()

    # Prediction on the test dataset - first five images
    Y_result = model.predict(test_images_cropped, verbose=1)
    Y_pred = np.argmax(Y_result, axis=1)

    #TODO - Application 1 - Step 8 - System evaluation - compute and display the
prediction error
    scores = model.evaluate(test_images_cropped, test_labels_cropped, verbose=2)
    print("Baseline Error: {:.2f}".format(100-scores[1]*100))
    return

######################################################################
def main():

    #TODO - Application 1 - Step 1 - Load the MNIST dataset in Keras
    (train_images, train_labels), (test_images, test_labels) = mnist.load_data()
    (train_images, train_labels), (test_images, test_labels) =
tf.keras.datasets.mnist.load_data()
    neurons=8
    batch_size=200
    mlp = PredictMLP(train_images, train_labels, test_images, test_labels, neurons,
batch_size)


    return
######################################################################

if __name__ == '__main__':
    main()
```

Figure 5: code for this part

The parts we have highlighted in yellow are the ones we have heavily modified to answer the exercise. We have reused the code given for the TP and adapted it in our particular case: after reshaping the images as it had to be done in Step 5 of the TP, we only used the five first images of the test dataset (instruction "`test_images_cropped = test_images[0:5]`") and their labels (instruction "`test_labels_cropped = test_labels[0:5]`"). We then create the model (instruction "`model = baseline_model(num_pixels, num_classes, neurons)`") and reload the weights (instruction "`model.load_weights('./application1.h5')`") with the help of the file created in Exercise 4.

The figure 6 shows the output we have obtained:

```
size of the initial test dataset (images) :10000
size of the initial test dataset (labels) :10000
size of the test dataset containing the 5 first images (images) :5
size of the test dataset containing the 5 first images :5
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 8)                 6280

 dense_1 (Dense)             (None, 10)                90

=================================================================
Total params: 6370 (24.88 KB)
Trainable params: 6370 (24.88 KB)
Non-trainable params: 0 (0.00 Byte)
_________________________________________________________________
1/1 [==============================] - 1s 628ms/step
1/1 - 1s - loss: 0.0157 - accuracy: 1.0000 - 546ms/epoch - 546ms/step
Baseline Error: 0.00
```

Figure 6: output for this part

We note that according to the accuracy and the baseline error, all the images have been classified properly. It seemed a bit strange to us to see this absence of flaws: it could be because our model has been tested on a very small amount of images (5), which, given its overall good results, did not allow us to see its potential mistakes and misclassifications.

I) Application II :

Please note: for your convenience, the outputs of this part have been grouped by exercise, not in one single part of the report.

The objective of this part is to develop a more sophisticated neural network (a Convolutional Neural Network) to classify the digits. The goal remains the same; in this part, only the neural network (that is to say, the classification "method") changes. We filled in the blanks left in the code of the first application. Please note that the names of the variables have been changed according to the table in part I (it helped us improve our understanding of the code) and that the code has been left in its"final" state (that is to say, its state when we solved the last exercise). If you want to check our solutions to the previous exercises, please uncomment the corresponding lines.

c) Code for this part

```
import keras
import tensorflow as tf
import time
from keras.datasets import mnist
#from keras.utils import np_utils
from keras.utils import to_categorical
from keras import layers
######################################################################
def baseline_model(num_pixels, num_classes, neurons):

    #TODO - Application 1 - Step 6a - Initialize the sequential model
    model = tf.keras.models.Sequential()    # Modify this

    #TODO - Application 1 - Step 6b - Define a hidden dense layer with 8 neurons
    model.add(layers.Dense(neurons, input_dim=num_pixels, kernel_initializer='normal',
activation='relu')) # the number of neurons has to be modified here
    # for exercise 1
```

```
    #TODO - Application 1 - Step 6c - Define the output dense layer
    model.add(layers.Dense(num_classes, kernel_initializer='normal',
activation='softmax'))

    # TODO - Application 1 - Step 6d - Compile the model
    model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])

    # the following line has been used to solve exercise 3:
    # model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['mse'])

    return model
########################################################################

def trainAndPredictMLP(train_images, train_labels, test_images, test_labels, neurons,
batch_size):

    #TODO - Application 1 - Step 3 - Reshape the MNIST dataset - Transform the images to
1D vectors of floats (28x28 pixels  to  784 elements)
    num_pixels = train_images.shape[1] * train_images.shape[2]
    train_images = train_images.reshape((train_images.shape[0],
num_pixels)).astype('float32')
    test_images = test_images.reshape((test_images.shape[0],
num_pixels)).astype('float32')

    #TODO - Application 1 - Step 4 - Normalize the input values
    train_images = train_images / 255
    test_images = test_images / 255

    #TODO - Application 1 - Step 5 - Transform the classes labels into a binary matrix
    train_labels = to_categorical(train_labels)
    test_labels = to_categorical(test_labels)
    num_classes = test_labels.shape[1]

    #TODO - Application 1 - Step 6 - Build the model architecture - Call the
baseline_model function

    model = baseline_model(num_pixels, num_classes, neurons)  #Modify this


    #TODO - Application 1 - Step 7 - Train the model
    start_time=time.time()
    model.fit(train_images, train_labels, validation_data=(test_images, test_labels),
epochs=10, batch_size=batch_size, verbose=2)
    end_time=time.time()

    # the following line has been used to solve exercise 4
    model.save_weights('./application1.h5')

    convergence_time=end_time-start_time
    #TODO - Application 1 - Step 8 - System evaluation - compute and display the
```

```
prediction error
    scores = model.evaluate(test_images, test_labels, verbose=2)
    print("Baseline Error: {:.2f}".format(100-scores[1]*100))
    print("the convergence time is", convergence_time, "seconds")

    return
#########################################################################

def CNN_model(input_shape, num_classes, size, neurons):

    # TODO - Application 2 - Step 5a - Initialize the sequential model
    model = tf.keras.models.Sequential()    #Modify this

    #TODO - Application 2 - Step 5b - Create the first hidden layer as a convolutional
layer
    # model.add(layers.Conv2D(8, size, activation='relu', input_shape=(28, 28, 1)))
    model.add(layers.Conv2D(30, size, activation='relu', input_shape=(28, 28, 1)))
    #the line above has been modified for exercise 9

    #TODO - Application 2 - Step 5c - Define the pooling layer
    model.add(layers.MaxPooling2D(pool_size=(2,2)))

    #the two following lines have been added in Exercise 9
    model.add(layers.Conv2D(15, (3,3), activation='relu'))
    model.add(layers.MaxPooling2D(pool_size=(2,2)))

    #TODO - Application 2 - Step 5d - Define the Dropout layer
    model.add(layers.Dropout(rate=0.2))

    #TODO - Application 2 - Step 5e - Define the flatten layer
    model.add(layers.Flatten())

    #TODO - Application 2 - Step 5f - Define a dense layer of size 128
    # model.add(layers.Dense(128, activation='relu'))
    model.add(layers.Dense(neurons, activation='relu'))

    #the following line has been added in Exercise 9
    model.add(layers.Dense(50, activation='relu'))

    #TODO - Application 2 - Step 5g - Define the output layer
    model.add(layers.Dense(num_classes, activation='softmax'))

    #TODO - Application 2 - Step 5h - Compile the model
    model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])

    return model
#########################################################################

def trainAndPredictCNN(train_images, train_labels, test_images, test_labels, size,
neurons, epochs):
```

```
    #TODO - Application 2 - Step 2 - reshape the data to be of size
[samples][width][height][channels]
    train_images = train_images.reshape((train_images.shape[0], 28, 28,
1)).astype('float32')
    test_images = test_images.reshape((test_images.shape[0], 28, 28,
1)).astype('float32')

    #TODO - Application 2 - Step 3 - normalize the input values from 0-255 to 0-1
    train_images = train_images / 255
    test_images = test_images / 255

    #TODO - Application 2 - Step 4 - One hot encoding - Transform the classes labels
into a binary matrix
    train_labels = to_categorical(train_labels)
    test_labels = to_categorical(test_labels)

    #TODO - Application 2 - Step 5 - Call the cnn_model function
    model = CNN_model(input_shape=(28,28,1), num_classes=10, size=size, neurons=neurons)
#Modify this


    #TODO - Application 2 - Step 6 - Train the model
    start_time=time.time()
    model.fit(train_images, train_labels, validation_data=(test_images, test_labels),
epochs=epochs, batch_size=200, verbose=1)
    end_time=time.time()

    convergence_time=end_time-start_time
    #TODO - Application 2 - Step 8 - Final evaluation of the model - compute and display
the prediction error
    scores = model.evaluate(test_images, test_labels, verbose=2)
    print("Baseline Error: {:.2f}".format(100-scores[1]*100))
    print("the convergence time is", convergence_time, "seconds")

    return
########################################################################

def main():

    #TODO - Application 1 - Step 1 - Load the MNIST dataset in Keras
    (train_images, train_labels), (test_images, test_labels) = mnist.load_data()

    #TODO - Application 1 - Step 2 - Train and predict on a MLP - Call the
trainAndPredictMLP function
    # loop used to solve exercise 1
    # neurons =[8,16,32,64,128]
    # for neuron in neurons :
    #    print("system performance with :", neuron,"neurons")
    #    mlp = trainAndPredictMLP(train_images, train_labels, test_images, test_labels,
neuron, batch_size=200)

    ### MLP : application 1, exercise 4
```

```
    # neurons=8
    # batch_size=200
    # mlp = trainAndPredictMLP(train_images, train_labels, test_images, test_labels,
neurons, batch_size)

    # loop used to solve exercise 2
    # batch_sizes =[32,64,128,256,512]
    # for batch_size in batch_sizes :
    #    print("system performance with the following batch size :", batch_size)
    #    mlp = trainAndPredictMLP(train_images, train_labels, test_images, test_labels,
neurons, batch_size)

    #TODO - Application 2 - Step 1 - Train and predict on a CNN - Call the
trainAndPredictCNN function

    #the following lines have been used to solve exercise 6
    # sizes = [(1,1), (3,3), (5,5), (7,7), (9,9)]
    # epochs = 10
    # neurons = 128
    # for size in sizes :
    #    print("system performance for the following size of the feature map of the
convolutional layer:", size)
    #    cnn = trainAndPredictCNN(train_images, train_labels, test_images, test_labels,
size, neurons, epochs)

    #the following lines have been used to solve exercise 7
    #neurons = [16, 64, 128, 256, 512]
    #size = (3,3)
    # epochs = 10
    #for neuron in neurons :
    #    print("system performance with :", neuron,"neurons")
    #    cnn = trainAndPredictCNN(train_images, train_labels, test_images, test_labels,
size, neuron, epochs)

    #the following lines have been used to solve exercise 8
    #size = (3,3)
    #neurons = 128
    #epochs = [1, 2, 5, 10, 20]
    #for epoch in epochs :
    #    print("system performance with :", epoch,"epochs")
    #    cnn = trainAndPredictCNN(train_images, train_labels, test_images, test_labels,
size, neurons, epoch)

    #the following lines have been used in Exercise 9
    size = (5,5)
    neurons = 128
    epochs = 10
    cnn = trainAndPredictCNN(train_images, train_labels, test_images, test_labels, size,
neurons, epochs)


    return
```

```
######################################################################

if __name__ == '__main__':
    main()
```

d) Answer for all the exercises related to the script, and outputs of the script related to each exercise

Exercise 6: Modify the size of the feature map within the convolutional layer as specified in Table 3. How is the system accuracy influenced by this parameter?

In this exercise, we are asked to modify the size of the feature map of the convolutional layer "Conv2D" and to perform the training of our model each time. We assessed the system accuracy for each size of the feature map.

To understand our implementation, let's take a look at the yellow part of the code above. It features the following loop (commented below):

```
    #the following lines have been used to solve exercise 6
    # sizes = [(1,1), (3,3), (5,5), (7,7), (9,9)]
    # epochs = 10
    # neurons = 128
    # for size in sizes :
    #    print("system performance for the following size of the feature map of the convolutional layer:", size)
    #    cnn = trainAndPredictCNN(train_images, train_labels, test_images, test_labels, size, neurons, epochs)
```

This loop has been used to perform the training of the different models in one run of our script. To do so, we defined an array of sizes called "`sizes`", that contains all the sizes of the feature map that we are going to study. We also added the parameters "`epochs`" and "`neurons`" according to the requirements of the lab for this exercise (we could have specified them as numbers in the functions, but we preferred this implementation that allowed us to perform loops in order to train the model for different epoch sizes and numbers of neurons, something that will be required in the next exercises). Then, the loop "`for`" allows us to train the model for each size of the feature map by calling the "`trainAndPredictCNN`" function. We obtained the following outputs and table of results:

```
system performance for the following size of the feature map of the convolutional layer: (1, 1)
Epoch 1/5
300/300 [==============================] - 13s 32ms/step - loss: 0.6387 - accuracy: 0.8356 - val_loss: 0.3148 - val_accuracy: 0.9091
Epoch 2/5
300/300 [==============================] - 7s 25ms/step - loss: 0.3284 - accuracy: 0.9003 - val_loss: 0.2543 - val_accuracy: 0.9243
Epoch 3/5
300/300 [==============================] - 7s 24ms/step - loss: 0.2732 - accuracy: 0.9168 - val_loss: 0.2074 - val_accuracy: 0.9389
Epoch 4/5
300/300 [==============================] - 8s 25ms/step - loss: 0.2256 - accuracy: 0.9308 - val_loss: 0.1739 - val_accuracy: 0.9495
Epoch 5/5
300/300 [==============================] - 9s 30ms/step - loss: 0.1869 - accuracy: 0.9425 - val_loss: 0.1434 - val_accuracy: 0.9562
313/313 - 3s - loss: 0.1434 - accuracy: 0.9562 - 3s/epoch - 11ms/step
Baseline Error: 4.38
the convergence time is 44.777148485183716 seconds
system performance for the following size of the feature map of the convolutional layer: (3, 3)
Epoch 1/5
300/300 [==============================] - 11s 31ms/step - loss: 0.3677 - accuracy: 0.8969 - val_loss: 0.1349 - val_accuracy: 0.9620
Epoch 2/5
300/300 [==============================] - 11s 38ms/step - loss: 0.1303 - accuracy: 0.9610 - val_loss: 0.0823 - val_accuracy: 0.9751
Epoch 3/5
300/300 [==============================] - 11s 36ms/step - loss: 0.0922 - accuracy: 0.9721 - val_loss: 0.0645 - val_accuracy: 0.9794
Epoch 4/5
300/300 [==============================] - 11s 37ms/step - loss: 0.0741 - accuracy: 0.9780 - val_loss: 0.0526 - val_accuracy: 0.9819
Epoch 5/5
300/300 [==============================] - 12s 39ms/step - loss: 0.0636 - accuracy: 0.9809 - val_loss: 0.0528 - val_accuracy: 0.9819
313/313 - 2s - loss: 0.0528 - accuracy: 0.9819 - 2s/epoch - 7ms/step
Baseline Error: 1.81
the convergence time is 55.72983407974243 seconds
```

```
system performance for the following size of the feature map of the convolutional layer: (5, 5)
Epoch 1/5
300/300 [==============================] - 14s 42ms/step - loss: 0.3724 - accuracy: 0.8916 - val_loss: 0.1413 - val_accuracy: 0.9588
Epoch 2/5
300/300 [==============================] - 13s 44ms/step - loss: 0.1348 - accuracy: 0.9597 - val_loss: 0.0791 - val_accuracy: 0.9755
Epoch 3/5
300/300 [==============================] - 13s 43ms/step - loss: 0.0935 - accuracy: 0.9712 - val_loss: 0.0573 - val_accuracy: 0.9820
Epoch 4/5
300/300 [==============================] - 12s 41ms/step - loss: 0.0712 - accuracy: 0.9779 - val_loss: 0.0484 - val_accuracy: 0.9842
Epoch 5/5
300/300 [==============================] - 12s 40ms/step - loss: 0.0583 - accuracy: 0.9819 - val_loss: 0.0438 - val_accuracy: 0.9851
313/313 - 3s - loss: 0.0438 - accuracy: 0.9851 - 3s/epoch - 11ms/step
Baseline Error: 1.49
the convergence time is 64.82084465026855 seconds
system performance for the following size of the feature map of the convolutional layer: (7, 7)
Epoch 1/5
300/300 [==============================] - 13s 42ms/step - loss: 0.3367 - accuracy: 0.9059 - val_loss: 0.1014 - val_accuracy: 0.9702
Epoch 2/5
300/300 [==============================] - 15s 49ms/step - loss: 0.1029 - accuracy: 0.9687 - val_loss: 0.0632 - val_accuracy: 0.9802
Epoch 3/5
300/300 [==============================] - 14s 47ms/step - loss: 0.0753 - accuracy: 0.9768 - val_loss: 0.0498 - val_accuracy: 0.9841
Epoch 4/5
300/300 [==============================] - 14s 45ms/step - loss: 0.0620 - accuracy: 0.9815 - val_loss: 0.0421 - val_accuracy: 0.9870
Epoch 5/5
300/300 [==============================] - 13s 45ms/step - loss: 0.0520 - accuracy: 0.9836 - val_loss: 0.0414 - val_accuracy: 0.9865
313/313 - 3s - loss: 0.0414 - accuracy: 0.9865 - 3s/epoch - 10ms/step
Baseline Error: 1.35
the convergence time is 69.36883354187012 seconds
```

```
system performance for the following size of the feature map of the convolutional layer: (9, 9)
Epoch 1/5
300/300 [==============================] - 15s 46ms/step - loss: 0.3608 - accuracy: 0.8949 - val_loss: 0.1247 - val_accuracy: 0.9625
Epoch 2/5
300/300 [==============================] - 16s 52ms/step - loss: 0.1190 - accuracy: 0.9652 - val_loss: 0.0669 - val_accuracy: 0.9782
Epoch 3/5
300/300 [==============================] - 16s 52ms/step - loss: 0.0831 - accuracy: 0.9747 - val_loss: 0.0533 - val_accuracy: 0.9829
Epoch 4/5
300/300 [==============================] - 15s 50ms/step - loss: 0.0658 - accuracy: 0.9799 - val_loss: 0.0475 - val_accuracy: 0.9847
Epoch 5/5
300/300 [==============================] - 15s 50ms/step - loss: 0.0552 - accuracy: 0.9830 - val_loss: 0.0399 - val_accuracy: 0.9873
313/313 - 3s - loss: 0.0399 - accuracy: 0.9873 - 3s/epoch - 8ms/step
Baseline Error: 1.27
the convergence time is 76.3290011882782 seconds
```

Size of the feature map	1 x 1	3 x 3	5 x 5	7 x 7	9 x 9
System Accuracy	0.9562	0.9819	0.9851	0.9865	0.9873
Convergence time (s)	44.8	55.7	64.8	69.4	76.3

Figure 7: Table of results for exercise 6

We notice that increasing the size of the feature map also increases the accuracy of our model. However, after a certain size, the accuracy of our system does not seem to increase a lot anymore (it seems "bounded"; for example, the accuracy does not increase so much between the size 3 x 3 and 5 x 5). This increase is caused by the fact that the feature maps are the results of filters that are multiplied with the input image. If the size of the filter is increased, it can recognize patterns more globally and in a better/more relevant way. However, increasing the size of the feature map seems to increase the convergence time as well: a tradeoff between accuracy and convergence time might need to be found to choose the "best" size of the feature map in a given context.

Exercise 7: Modify the number of neurons in the dense hidden layer as specified in Table 4. Select a value of 3x3 neurons for the convolutional kernel. What can be observed regarding the system accuracy? How about the convergence time necessary for the system to train a model?

In this exercise, we are asked to modify the number of neurons in the dense hidden layer. To do so, we implemented the following loop (highlighted in pink in the code above and commented here):

```
    #the following lines have been used to solve exercise 7
    #neurons = [16, 64, 128, 256, 512]
    #size = (3,3)
    # epochs = 10
    #for neuron in neurons :
    #    print("system performance with :", neuron,"neurons")
    #    cnn = trainAndPredictCNN(train_images, train_labels, test_images, test_labels,
size, neuron, epochs)
```

This loop has been used to perform the training of the different models in one run of our script. To do so, we defined an array of neurons containing the number of neurons, called "`neurons`". The loop "`for`" allowed us to train the model for each number of neurons that we wanted to study by calling the "`trainAndPredictCNN`" function. We obtained the following outputs and table of results:

```
system performance with : 16 neurons
Epoch 1/5
300/300 [==============================] - 15s 42ms/step - loss: 0.6000 - accuracy: 0.8264 - val_loss: 0.2336 - val_accuracy: 0.9368
Epoch 2/5
300/300 [==============================] - 10s 32ms/step - loss: 0.2238 - accuracy: 0.9359 - val_loss: 0.1570 - val_accuracy: 0.9554
Epoch 3/5
300/300 [==============================] - 8s 27ms/step - loss: 0.1706 - accuracy: 0.9498 - val_loss: 0.1234 - val_accuracy: 0.9651
Epoch 4/5
300/300 [==============================] - 9s 29ms/step - loss: 0.1466 - accuracy: 0.9567 - val_loss: 0.1055 - val_accuracy: 0.9687
Epoch 5/5
300/300 [==============================] - 10s 35ms/step - loss: 0.1303 - accuracy: 0.9613 - val_loss: 0.0972 - val_accuracy: 0.9712
313/313 - 2s - loss: 0.0972 - accuracy: 0.9712 - 2s/epoch - 7ms/step
Baseline Error: 2.88
the convergence time is 52.43540716171265 seconds
system performance with : 64 neurons
Epoch 1/5
300/300 [==============================] - 11s 33ms/step - loss: 0.4135 - accuracy: 0.8825 - val_loss: 0.1672 - val_accuracy: 0.9508
Epoch 2/5
300/300 [==============================] - 11s 37ms/step - loss: 0.1644 - accuracy: 0.9522 - val_loss: 0.1078 - val_accuracy: 0.9685
Epoch 3/5
300/300 [==============================] - 11s 35ms/step - loss: 0.1218 - accuracy: 0.9635 - val_loss: 0.0878 - val_accuracy: 0.9722
Epoch 4/5
300/300 [==============================] - 11s 36ms/step - loss: 0.1028 - accuracy: 0.9693 - val_loss: 0.0732 - val_accuracy: 0.9782
Epoch 5/5
300/300 [==============================] - 11s 36ms/step - loss: 0.0887 - accuracy: 0.9738 - val_loss: 0.0653 - val_accuracy: 0.9796
313/313 - 3s - loss: 0.0653 - accuracy: 0.9796 - 3s/epoch - 8ms/step
Baseline Error: 2.04
the convergence time is 54.035040616989136 seconds
```

```
system performance with : 128 neurons
Epoch 1/5
300/300 [==============================] - 11s 34ms/step - loss: 0.4315 - accuracy: 0.8783 - val_loss: 0.1783 - val_accuracy: 0.9484
Epoch 2/5
300/300 [==============================] - 11s 37ms/step - loss: 0.1620 - accuracy: 0.9521 - val_loss: 0.1036 - val_accuracy: 0.9693
Epoch 3/5
300/300 [==============================] - 11s 37ms/step - loss: 0.1154 - accuracy: 0.9653 - val_loss: 0.0810 - val_accuracy: 0.9752
Epoch 4/5
300/300 [==============================] - 11s 38ms/step - loss: 0.0917 - accuracy: 0.9731 - val_loss: 0.0680 - val_accuracy: 0.9780
Epoch 5/5
300/300 [==============================] - 11s 37ms/step - loss: 0.0786 - accuracy: 0.9756 - val_loss: 0.0636 - val_accuracy: 0.9794
313/313 - 3s - loss: 0.0636 - accuracy: 0.9794 - 3s/epoch - 8ms/step
Baseline Error: 2.06
the convergence time is 56.2769672870636 seconds
system performance with : 256 neurons
Epoch 1/5
300/300 [==============================] - 13s 40ms/step - loss: 0.3237 - accuracy: 0.9058 - val_loss: 0.1192 - val_accuracy: 0.9646
Epoch 2/5
300/300 [==============================] - 13s 44ms/step - loss: 0.1217 - accuracy: 0.9632 - val_loss: 0.0772 - val_accuracy: 0.9764
Epoch 3/5
300/300 [==============================] - 14s 46ms/step - loss: 0.0873 - accuracy: 0.9731 - val_loss: 0.0641 - val_accuracy: 0.9799
Epoch 4/5
300/300 [==============================] - 14s 47ms/step - loss: 0.0708 - accuracy: 0.9782 - val_loss: 0.0591 - val_accuracy: 0.9800
Epoch 5/5
300/300 [==============================] - 13s 43ms/step - loss: 0.0593 - accuracy: 0.9815 - val_loss: 0.0565 - val_accuracy: 0.9814
313/313 - 2s - loss: 0.0565 - accuracy: 0.9814 - 2s/epoch - 6ms/step
Baseline Error: 1.86
the convergence time is 67.21643209457397 seconds
```

```
system performance with : 512 neurons
Epoch 1/5
300/300 [==============================] - 16s 51ms/step - loss: 0.2945 - accuracy: 0.9190 - val_loss: 0.1036 - val_accuracy: 0.9694
Epoch 2/5
300/300 [==============================] - 16s 53ms/step - loss: 0.1087 - accuracy: 0.9671 - val_loss: 0.0689 - val_accuracy: 0.9788
Epoch 3/5
300/300 [==============================] - 16s 52ms/step - loss: 0.0767 - accuracy: 0.9770 - val_loss: 0.0560 - val_accuracy: 0.9818
Epoch 4/5
300/300 [==============================] - 16s 53ms/step - loss: 0.0600 - accuracy: 0.9815 - val_loss: 0.0486 - val_accuracy: 0.9838
Epoch 5/5
300/300 [==============================] - 16s 52ms/step - loss: 0.0500 - accuracy: 0.9841 - val_loss: 0.0467 - val_accuracy: 0.9853
313/313 - 4s - loss: 0.0467 - accuracy: 0.9853 - 4s/epoch - 13ms/step
Baseline Error: 1.47
the convergence time is 79.67010450363159 seconds
```

Number of neurons	16	64	128	256	512
System Accuracy	0.9712	0.9796	0.9794	0.9814	0.9853
Convergence time (s)	52.4	54.0	56.3	67.2	79.7

Figure 8 : Table of results for exercise 7

As we saw for the previous implementation of the classifier (in application 1), increasing the number of neurons increases the accuracy of our system, because it allows it to learn more complex patterns and to fit the training data in a better way. However, it also increases the convergence time. The previous implementation converges more quickly than the CNN, but it yielded results that were less satisfying in terms of accuracy (please see the table in exercise 1). Thus, we can see how efficient the CNN architecture is compared to the other implementation we have studied in this lab.

Exercise 8: Modify the number of epochs used to train the model as specified in Table 5. Select a value of 128 neurons in the dense hidden layer. What can be observed regarding the system accuracy? How about the convergence time?

Here, we are asked to modify the number of epochs used to train the model. We used the same strategy as in the exercises above, which is to say, we implemented the following loop (highlighted in blue in the code above and commented here):

```
    #the following lines have been used to solve exercise 8
    #size = (3,3)
    #neurons = 128
    #epochs = [1, 2, 5, 10, 20]
    #for epoch in epochs :
    #    print("system performance with :", epoch,"epochs")
    #    cnn = trainAndPredictCNN(train_images, train_labels, test_images, test_labels,
size, neurons, epoch)
```

This loop has been used to perform the training of the different models in one run of our script. To do so, we defined an array of epochs containing the number of epochs, called "`epoch`". The loop "`for`" allowed us to train the model for each number of epochs that we wanted to study by calling the "`trainAndPredictCNN`" function. We obtained the following outputs and table of results:

```
system performance with : 1 epochs
300/300 [==============================] - 16s 45ms/step - loss: 0.3634 - accuracy: 0.8943 - val_loss: 0.1589 - val_accuracy: 0.9553
313/313 - 4s - loss: 0.1589 - accuracy: 0.9553 - 4s/epoch - 13ms/step
Baseline Error: 4.47
the convergence time is 17.242071628570557 seconds
system performance with : 2 epochs
Epoch 1/2
300/300 [==============================] - 12s 33ms/step - loss: 0.3831 - accuracy: 0.8923 - val_loss: 0.1614 - val_accuracy: 0.9561
Epoch 2/2
300/300 [==============================] - 11s 37ms/step - loss: 0.1460 - accuracy: 0.9583 - val_loss: 0.0981 - val_accuracy: 0.9717
313/313 - 4s - loss: 0.0981 - accuracy: 0.9717 - 4s/epoch - 12ms/step
Baseline Error: 2.83
the convergence time is 23.2792649269104 seconds
system performance with : 5 epochs
Epoch 1/5
300/300 [==============================] - 13s 37ms/step - loss: 0.4058 - accuracy: 0.8912 - val_loss: 0.1577 - val_accuracy: 0.9559
Epoch 2/5
300/300 [==============================] - 12s 39ms/step - loss: 0.1518 - accuracy: 0.9544 - val_loss: 0.1008 - val_accuracy: 0.9696
Epoch 3/5
300/300 [==============================] - 12s 41ms/step - loss: 0.1086 - accuracy: 0.9675 - val_loss: 0.0752 - val_accuracy: 0.9763
Epoch 4/5
300/300 [==============================] - 12s 40ms/step - loss: 0.0880 - accuracy: 0.9735 - val_loss: 0.0647 - val_accuracy: 0.9791
Epoch 5/5
300/300 [==============================] - 12s 40ms/step - loss: 0.0756 - accuracy: 0.9768 - val_loss: 0.0581 - val_accuracy: 0.9802
313/313 - 2s - loss: 0.0581 - accuracy: 0.9802 - 2s/epoch - 7ms/step
Baseline Error: 1.98
the convergence time is 61.107154846191406 seconds
```

```
system performance with : 10 epochs
Epoch 1/10
300/300 [==============================] - 14s 39ms/step - loss: 0.4079 - accuracy: 0.8838 - val_loss: 0.1494 - val_accuracy: 0.9580
Epoch 2/10
300/300 [==============================] - 12s 39ms/step - loss: 0.1480 - accuracy: 0.9567 - val_loss: 0.0893 - val_accuracy: 0.9745
Epoch 3/10
300/300 [==============================] - 12s 42ms/step - loss: 0.1084 - accuracy: 0.9675 - val_loss: 0.0705 - val_accuracy: 0.9774
Epoch 4/10
300/300 [==============================] - 12s 40ms/step - loss: 0.0875 - accuracy: 0.9730 - val_loss: 0.0612 - val_accuracy: 0.9798
Epoch 5/10
300/300 [==============================] - 12s 40ms/step - loss: 0.0735 - accuracy: 0.9771 - val_loss: 0.0550 - val_accuracy: 0.9830
Epoch 6/10
300/300 [==============================] - 12s 40ms/step - loss: 0.0637 - accuracy: 0.9802 - val_loss: 0.0498 - val_accuracy: 0.9834
Epoch 7/10
300/300 [==============================] - 12s 41ms/step - loss: 0.0575 - accuracy: 0.9818 - val_loss: 0.0485 - val_accuracy: 0.9828
Epoch 8/10
300/300 [==============================] - 12s 40ms/step - loss: 0.0497 - accuracy: 0.9850 - val_loss: 0.0468 - val_accuracy: 0.9835
Epoch 9/10
300/300 [==============================] - 12s 40ms/step - loss: 0.0448 - accuracy: 0.9861 - val_loss: 0.0415 - val_accuracy: 0.9864
Epoch 10/10
300/300 [==============================] - 11s 38ms/step - loss: 0.0411 - accuracy: 0.9866 - val_loss: 0.0424 - val_accuracy: 0.9861
313/313 - 3s - loss: 0.0424 - accuracy: 0.9861 - 3s/epoch - 10ms/step
Baseline Error: 1.39
the convergence time is 122.27370858192444 seconds
```

```
system performance with : 20 epochs
Epoch 1/20
300/300 [==============================] - 12s 34ms/step - loss: 0.3982 - accuracy: 0.8889 - val_loss: 0.1610 - val_accuracy: 0.9541
Epoch 2/20
300/300 [==============================] - 11s 38ms/step - loss: 0.1518 - accuracy: 0.9554 - val_loss: 0.1005 - val_accuracy: 0.9692
Epoch 3/20
300/300 [==============================] - 11s 37ms/step - loss: 0.1099 - accuracy: 0.9665 - val_loss: 0.0805 - val_accuracy: 0.9755
Epoch 4/20
300/300 [==============================] - 11s 38ms/step - loss: 0.0857 - accuracy: 0.9744 - val_loss: 0.0628 - val_accuracy: 0.9797
Epoch 5/20
300/300 [==============================] - 11s 37ms/step - loss: 0.0744 - accuracy: 0.9775 - val_loss: 0.0550 - val_accuracy: 0.9818
Epoch 6/20
300/300 [==============================] - 11s 37ms/step - loss: 0.0638 - accuracy: 0.9806 - val_loss: 0.0557 - val_accuracy: 0.9810
Epoch 7/20
300/300 [==============================] - 11s 37ms/step - loss: 0.0584 - accuracy: 0.9820 - val_loss: 0.0523 - val_accuracy: 0.9814
Epoch 8/20
300/300 [==============================] - 11s 37ms/step - loss: 0.0501 - accuracy: 0.9845 - val_loss: 0.0456 - val_accuracy: 0.9837
Epoch 9/20
300/300 [==============================] - 12s 39ms/step - loss: 0.0466 - accuracy: 0.9854 - val_loss: 0.0443 - val_accuracy: 0.9858
Epoch 10/20
300/300 [==============================] - 11s 38ms/step - loss: 0.0409 - accuracy: 0.9871 - val_loss: 0.0411 - val_accuracy: 0.9863
Epoch 11/20
300/300 [==============================] - 16s 52ms/step - loss: 0.0377 - accuracy: 0.9882 - val_loss: 0.0432 - val_accuracy: 0.9850
Epoch 12/20
300/300 [==============================] - 14s 48ms/step - loss: 0.0348 - accuracy: 0.9887 - val_loss: 0.0412 - val_accuracy: 0.9869
Epoch 13/20
300/300 [==============================] - 18s 59ms/step - loss: 0.0314 - accuracy: 0.9901 - val_loss: 0.0415 - val_accuracy: 0.9857
Epoch 14/20
300/300 [==============================] - 16s 52ms/step - loss: 0.0292 - accuracy: 0.9909 - val_loss: 0.0415 - val_accuracy: 0.9848
```

```
Epoch 15/20
300/300 [==============================] - 14s 48ms/step - loss: 0.0256 - accuracy: 0.9916 - val_loss: 0.0413 - val_accuracy: 0.9864
Epoch 16/20
300/300 [==============================] - 11s 37ms/step - loss: 0.0239 - accuracy: 0.9922 - val_loss: 0.0405 - val_accuracy: 0.9873
Epoch 17/20
300/300 [==============================] - 11s 37ms/step - loss: 0.0218 - accuracy: 0.9927 - val_loss: 0.0425 - val_accuracy: 0.9862
Epoch 18/20
300/300 [==============================] - 11s 38ms/step - loss: 0.0188 - accuracy: 0.9941 - val_loss: 0.0392 - val_accuracy: 0.9876
Epoch 19/20
300/300 [==============================] - 11s 37ms/step - loss: 0.0185 - accuracy: 0.9942 - val_loss: 0.0423 - val_accuracy: 0.9877
Epoch 20/20
300/300 [==============================] - 11s 37ms/step - loss: 0.0175 - accuracy: 0.9944 - val_loss: 0.0405 - val_accuracy: 0.9876
313/313 - 2s - loss: 0.0405 - accuracy: 0.9876 - 2s/epoch - 7ms/step
Baseline Error: 1.24
the convergence time is 246.78136801719666 seconds
```

Number of epochs	1	2	5	10	20
System Accuracy	0.9553	0.9717	0.9802	0.9861	0.9876
Convergence time (s)	17.2	23.3	61.1	122.3	246.8

Figure 9 : Table of results for exercise 8

Increasing the number of epochs used to train the model increases the accuracy. An epoch is defined as one pass of the training set in the model (it is the number of times the model "sees" the training data and learns from it). If the model does this several times, it allows it to learn better, thus increasing the accuracy. However, the increase does not seem to be significant anymore after a certain number of epochs (it does not seem to increase so much between 10 and 20 epochs. However, the convergence time doubles! It is why a tradeoff has to be found between increasing the accuracy and increasing the number of epochs, which might make the model converge only after a lot of time.) This increase of the convergence time can be explained by the fact that "seeing the data several times" requires more computations, which take time.

Exercise 9: Modify the CNN architecture with additional convolutional, max pooling layers and fully connected layers

In this last exercise, we added (or modified) the lines in purple in the code above, to add layers to our CNN architecture. We then determined the accuracy of the system, given by the following output:

```
Epoch 1/10
300/300 [==============================] - 23s 70ms/step - loss: 0.3814 - accuracy: 0.8841 - val_loss: 0.0750 - val_accuracy: 0.9767
Epoch 2/10
300/300 [==============================] - 26s 86ms/step - loss: 0.0937 - accuracy: 0.9712 - val_loss: 0.0491 - val_accuracy: 0.9830
Epoch 3/10
300/300 [==============================] - 26s 87ms/step - loss: 0.0718 - accuracy: 0.9777 - val_loss: 0.0397 - val_accuracy: 0.9869
Epoch 4/10
300/300 [==============================] - 26s 87ms/step - loss: 0.0594 - accuracy: 0.9819 - val_loss: 0.0346 - val_accuracy: 0.9894
Epoch 5/10
300/300 [==============================] - 28s 92ms/step - loss: 0.0494 - accuracy: 0.9844 - val_loss: 0.0321 - val_accuracy: 0.9882
Epoch 6/10
300/300 [==============================] - 30s 99ms/step - loss: 0.0445 - accuracy: 0.9859 - val_loss: 0.0263 - val_accuracy: 0.9922
Epoch 7/10
300/300 [==============================] - 26s 86ms/step - loss: 0.0375 - accuracy: 0.9879 - val_loss: 0.0248 - val_accuracy: 0.9912
Epoch 8/10
300/300 [==============================] - 26s 86ms/step - loss: 0.0371 - accuracy: 0.9883 - val_loss: 0.0253 - val_accuracy: 0.9918
Epoch 9/10
300/300 [==============================] - 26s 85ms/step - loss: 0.0319 - accuracy: 0.9899 - val_loss: 0.0234 - val_accuracy: 0.9919
Epoch 10/10
300/300 [==============================] - 28s 94ms/step - loss: 0.0298 - accuracy: 0.9901 - val_loss: 0.0236 - val_accuracy: 0.9917
313/313 - 4s - loss: 0.0236 - accuracy: 0.9917 - 4s/epoch - 12ms/step
Baseline Error: 0.83
the convergence time is 263.44080209732056 seconds
```

We chose to restore the parameters used in the previous exercises (namely, we performed the training for 10 epochs). The accuracy of our system has reached 99.2%: adding layers, or slightly altering the architecture has drastically improved it. Even if the convergence time is quite long, the model's performance is relatively good. Adding more convolutional layers seems to allow the model to learn more complex patterns, whereas adding fully connected "dense" layers seems to allow the model to generalize them more easily, thus leading to an increase in accuracy.