Lab Report TP-IMA4103
TP 2: Fashion Classification

Yakine KLABI

Outline :

Introduction :

The objective of this lab is to make us use CNNs in order to classify clothes. We will develop from scratch a CNN that is able to recognize an item of clothing on an input image. To do so, we will use Tensorflow with the Keras API. The first application will be focused on classifying items from the following categories: 0 - T-shirt/top, 1 - Trouser, 2 - Pullover, 3 - Dress, 4 - Coat, 5 - Sandal, 6 - Shirt, 7 - Sneaker, 8 - Bag and 9 - Ankle boot. The evaluation will be performed on the testing dataset (10.000 images), whereas in Application 2, the training strategy will be changed. Indeed, three datasets will be used: one for training, one for validation and the final one for testing purposes. We will compare those two strategies in terms of accuracy and loss, to evaluate the performance of our model.

I) Application I :

The objective of this part is to write a Python script that is able to recognize the category of an unknown image applied as input. The image contains an item of clothing. To achieve this objective, we will use a dataset named “Fashion-MNIST”. The inputs contained images of clothes, shoes and bags. Thus, the mapping classes labels (which are integers from 0 to 9) can be defined as follows : 0 - T-shirt/top, 1 - Trouser, 2 - Pullover, 3 - Dress, 4 - Coat, 5 - Sandal, 6 - Shirt, 7 - Sneaker, 8 - Bag and 9 - Ankle boot.

This section contains the code for all the exercises. The parts corresponding to exercises are commented and highlighted in a different color. Then, we will display the outputs of each exercise.

a) The code of the whole application:

```
# Application 1 - Step 1 - Import the dependencies
import numpy as np
from sklearn.model_selection import KFold
import keras
from keras.optimizers import SGD
from keras.datasets import fashion_mnist
from keras.utils import to_categorical
from keras import layers
from keras.layers import Dropout
from matplotlib import pyplot
import cv2
import time
#############################################################################################
def summarizeLearningCurvesPerformances(histories, accuracyScores):

    for i in range(len(histories)):
        # plot loss
        pyplot.subplot(211)
        pyplot.title('Cross Entropy Loss')
        pyplot.plot(histories[i].history['loss'], color='green', label='train')
        pyplot.plot(histories[i].history['val_loss'], color='red', label='test')

        # plot accuracy
        pyplot.subplot(212)
        pyplot.title('Classification Accuracy')
        pyplot.plot(histories[i].history['accuracy'], color='green', label='train')
        pyplot.plot(histories[i].history['val_accuracy'], color='red', label='test')

        #print accuracy for each split
        print("Accuracy for set {} = {}".format(i, accuracyScores[i]))

    pyplot.show()

    print('Accuracy: mean = {:.3f} std = {:.3f}, n = {}'.format(np.mean(accuracyScores) * 100,
np.std(accuracyScores) * 100, len(accuracyScores)))
#############################################################################################


def prepareData(trainX, trainY, testX, testY):
```

```
    #TODO - Application 1 - Step 4a - reshape the data to be of size
[samples][width][height][channels]
    trainX = trainX.reshape((trainX.shape[0], 28, 28, 1))
    testX = testX.reshape((testX.shape[0], 28, 28, 1))

    #TODO - Application 1 - Step 4b - normalize the input values
    trainX = trainX.astype('float32') / 255
    testX = testX.astype('float32') / 255

    #TODO - Application 1 - Step 4c - Transform the classes labels into a binary matrix
    trainY = to_categorical(trainY)
    testY = to_categorical(testY)

    return trainX, trainY, testX, testY
#################################################################################################
###########################################################################
#def defineModel(input_shape, num_classes, num_filters):#used for exercise 2
#def defineModel(input_shape, num_classes):used for exercise 3
#def defineModel(input_shape, num_classes, learning_rate):used for exercise 5
#def defineModel(input_shape, num_classes, dropout_percentage):used for exercise 6

def defineModel(input_shape, num_classes):

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

    #TODO - Application 1 - Step 6b - Create the first hidden layer as a convolutional layer
    model.add(layers.Conv2D(64, (3, 3), activation='relu', kernel_initializer='he_uniform',
input_shape=input_shape))#use the optimal value of filters number

    #model.add(layers.Conv2D(num_filters, (3, 3), activation='relu',
kernel_initializer='he_uniform', input_shape=input_shape))#used for exercise2

    #TODO - Application 1 - Step 6c - Define the pooling layer
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(Dropout(0.2))

    #model.add(Dropout(dropout_percentage)): used for exercise 6

    #TODO - Application 1 - Step 6d - Define the flatten layer
    model.add(layers.Flatten())

    #TODO - Application 1 - Step 6e - Define a dense layer of size 16
    model.add(layers.Dense(128, activation='relu', kernel_initializer='he_uniform')) #use the
optimal value of neurons number
    #model.add(layers.Dense(neurons, activation='relu', kernel_initializer='he_uniform'))

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

    #TODO - Application 1 - Step 6g - Compile the model
```

```
    model.compile(optimizer=SGD(learning_rate=0.01, momentum=0.9),
loss='categorical_crossentropy', metrics=['accuracy'])

    return model
###############################################################################################
###################Loading image for exercise 8########################################

def load_image(filename):
    img = cv2.imread(filename, cv2.IMREAD_GRAYSCALE)
    img = cv2.resize(img, (28, 28))
    img = img.astype('float32') / 255.0
    img = img.reshape(1, 28, 28, 1)
    return img

##################################################################################
#def defineTrainAndEvaluateClassic(trainX, trainY, testX, testY, neurons):used for exercise 3
#def defineTrainAndEvaluateClassic(trainX, trainY, testX, testY, epoch):used for exercise 4
#def defineTrainAndEvaluateClassic(trainX, trainY, testX, testY, learning_rate):used for exercise
5
#def defineTrainAndEvaluateClassic(trainX, trainY, testX, testY, droupout_percentage):used for
exercise 6
def defineTrainAndEvaluateClassic(trainX, trainY, testX, testY):

    #TODO - Application 1 - Step 6 - Call the defineModel function
    #start of training : used to calculate the training time
    #start_time=time.time()

    #model = defineModel((28, 28, 1), 10, num_filters): used for exercise 2
    #model = defineModel((28, 28, 1), 10) #used for exercise 3
    #model = defineModel((28, 28, 1), 10, droupout_percentage) : used for exercise 6

    model = defineModel((28, 28, 1), 10) #used to train with optimal values

######################### used for exercise 2 and 3 ##########################
############################################################################""
    #this code is used to solve the second exercise, to calculate the convergence time and the
system accuracy with different filters
    #epochs = 5
    #prev_accuracy = 0
    #start_time = time.time()
    #for epoch in range(epochs):
        #history = model.fit(trainX, trainY, epoch=1, batch_size=32, validation_data=(testX,
testY), verbose=1)
        #_, accuracy = model.evaluate(testX, testY, verbose=0)
        #if np.abs(accuracy - prev_accuracy) < 0.001:  # Convergence criteria
         #   break
        #prev_accuracy = accuracy
##################################################################################"
    #TODO - Application 1 - Step 7 - Train the model
    model.fit(trainX, trainY, epochs=10, batch_size=32, validation_data=(testX, testY),
verbose=1) #change to optimal value of epoch to 10
    #TODO - Application 1 - Step 8 - Evaluate the model
```

```
    loss, accuracy = model.evaluate(testX, testY, verbose=0)

    #used to calculate the training time for exercise 1
    #end_time=time.time()

    #print('Number of Filters: {}'.format(num_filters)) : used for exercise2
    #print('Number of neurons: {}'.format(neurons))  #used for exercise3
#print('Train the model with :', epoch, 'epoch')#used for exercise4
    #print('Train the model with :', learning_rate, 'as a learning rate')#used for exercise5
    #print('Train the model with :', droupout_percentage, 'as a dropout percentage')#used for
exercise 6

    print('Test Accuracy: %.3f' % (accuracy * 100.0))

    #convergence_time = end_time-start_time
    #print("Convergence Time: {:.2f} seconds".format(convergence_time))

################################################################################""
###this part is used to print the training time in exercise 1###
    #print("Training Time (CPU): {:.2f} seconds".format(end_time - start_time))

    return model
    return
##################################################################################
def defineTrainAndEvaluateKFolds(trainX, trainY, testX, testY):

    k_folds = 5

    accuracyScores = []
    histories = []

    #Application 2 - Step 2 - Prepare the cross validation datasets
    kfold = KFold(k_folds, shuffle=True, random_state=1)

    for train_idx, val_idx in kfold.split(trainX):

        #TODO - Application 2 - Step 3 - Select data for train and validation


        #TODO - Application 2 - Step 4 - Build the model - Call the defineModel function


        #TODO - Application 2 - Step 5 - Fit the model


        #TODO - Application 2 - Step 6 - Save the training related information in the histories
list


        #TODO - Application 2 - Step 7 - Evaluate the model on the test dataset
```

```python
        #TODO - Application 2 - Step 8 - Save the accuracy in the accuracyScores list

        pass #DELETE THIS!

    return histories, accuracyScores
#####################################################################################
def main():

    #TODO - Application 1 - Step 2 - Load the Fashion MNIST dataset in Keras
    (trainX, trainY), (testX, testY) = fashion_mnist.load_data()

    #TODO - Application 1 - Step 3 - Print the size of the train/test dataset
    print('Train:', trainX.shape, trainY.shape)
    print('Test:', testX.shape, testY.shape)


    #TODO - Application 1 - Step 4 - Call the prepareData method
    trainX, trainY, testX, testY = prepareData(trainX, trainY, testX, testY)

    #TODO - Application 1 - Step 5 - Define, train and evaluate the model in the classical way
    defineTrainAndEvaluateClassic(trainX, trainY, testX, testY)
     model.save('./Fashion_MNIST_model.h5')  #save the model for exercise 8


    #######################a loop used to solve the exercise 2 in order to change the number of filter########################"
    #num_filters_list = [8, 16, 32, 64, 128]
    #for num_filters in num_filters_list:
        #defineTrainAndEvaluateClassic(trainX, trainY, testX, testY, num_filters)
    ###########################################################################################

    #######################a loop used to solve the exercise 3 in order to change the number of neurons########################
    #neurons = [16, 64, 128, 256, 512]
    #for neuron in neurons:
     #   defineTrainAndEvaluateClassic(trainX, trainY, testX, testY, neuron)
    ###########################################################################################

    #######################a loop used to solve the exercise 4 in order to change the number of epochs########################
    #epochs = [1, 2, 5, 10, 20]
    #for epoch in epochs:
     #   defineTrainAndEvaluateClassic(trainX, trainY, testX, testY, epoch)
    ###########################################################################################


    #######################a loop used to solve the exercise 5 in order to change the learning rate of SGD optimize########################
    #learning_rates = [0.1, 0.01, 0.001, 0.0001, 0.00001]
    #for learning_rate in learning_rates:
     #   defineTrainAndEvaluateClassic(trainX, trainY, testX, testY, learning_rate)
    ###########################################################################################
```

```
    ########################a loop used to solve the exercise 6 in order to change the dropout percentage########################
    #dropouts = [0.1, 0.2, 0.3, 0.4, 0.5]
    #for dropout in dropouts:
     #   defineTrainAndEvaluateClassic(trainX, trainY, testX, testY, dropout)

#################################################################################################################################
#######################This part responds to exercise 8 ####################################

#Load the pretrained model
model=load_model('./Fashion_MNIST_model.h5')

#load the query image using the method above
query_img = load_image('sample_image.png')

#generate predictions for the query image and return probabilities for each class
prediction = model.predict(query_img)

#used to find the index of the highest value in the prediction array
predicted_class = np.argmax(prediction)

#define the classes labels

classes_labels= {0: 'T-shirt/top', 1: 'Trouser', 2: 'Pullover', 3: 'Dress', 4: 'Coat', 5: 'Sandal', 6: 'Shirt', 7: 'Sneaker', 8: 'Bag', 9: 'Ankle boot'}

#assigns the predicted class to its category

category= classes_labels[predicted_class]

print('Predicted Category for the query image is :', category)
##########################################################################################################################
    #TODO - Application 2 - Step 1 - Define, train and evaluate the model using K-Folds strategy

    #TODO - Application 2 - Step9 - System performance presentation
    return
#####################################################################
if __name__ == '__main__':
    main()
#####################################################################
```

b) ***Exercise 1 : Take a look at the gray part and the steps of the code above***

This exercise corresponds to all the steps from 1 to 9. Below, you will find the output of the implemented code and the result of the training time when we run the script on a CPU and on a GPU.

The results below show that the model achieved a test accuracy of 89.51%. This means that when presented with unseen data (the test set), the model correctly classified approximately 89.51% of the samples. Regarding the training time, it took approximately 24.46 seconds to train the model on a CPU. This duration represents the total time taken to complete all epochs of training.

```
1627/1875 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.9167
1637/1875 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.9167
1647/1875 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.9167
1657/1875 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.9167
1667/1875 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.9167
1677/1875 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.9166
1687/1875 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.9166
1697/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.91661707/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.91661716/1875 ━━━━
1707/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9166━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.91661737/1875 ━━━━━━━━━━━━━━━━━━━━
1716/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9166━━ 0s 5ms/step - accuracy: 0.91651756/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step
1726/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9166 - accuracy: 0.91651774/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0
1737/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9165.91651792/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.91651801/1875
1746/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9165 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.91641820/1875 ━━━━━━━━━━━━━━
1756/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9165━━━━━━ 0s 5ms/step - accuracy: 0.91641841/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms
1766/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9165/step - accuracy: 0.91641860/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accura
1774/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9165cy: 0.91631875/1875 ━━━━━━━━━━━━━━━━━━━━ 11s 6ms/step - accuracy: 0.9163 -
1783/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9165
1792/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9165
1801/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9164
1811/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9164fication_Moodle\TP2_MNIST_Fashion_Classification_Moodle>
1820/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9164
1830/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9164
1841/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9164
1851/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9164
1860/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9164
1869/1875 ━━━━━━━━━━━━━━━━━━━━ 0s 5ms/step - accuracy: 0.9163
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 11s 6ms/step - accuracy: 0.916
3 - loss: 0.2343 - val_accuracy: 0.8962 - val_loss: 0.2974
Test Accuracy: 89.620
Training Time (CPU): 47.73 seconds
```


Figure 1 : The output of the training on CPU

```
Train: (60000, 28, 28) (60000,)
Test: (10000, 28, 28) (10000,)
Epoch 1/5
1875/1875 [==============================] - 7s 3ms/step - loss: 0.4613 - accuracy: 0.8329 - val_loss: 0.3705 - val_accuracy: 0.8680
Epoch 2/5
1875/1875 [==============================] - 7s 4ms/step - loss: 0.3112 - accuracy: 0.8902 - val_loss: 0.3238 - val_accuracy: 0.8844
Epoch 3/5
1875/1875 [==============================] - 6s 3ms/step - loss: 0.2752 - accuracy: 0.9013 - val_loss: 0.2941 - val_accuracy: 0.8928
Epoch 4/5
1875/1875 [==============================] - 7s 4ms/step - loss: 0.2527 - accuracy: 0.9092 - val_loss: 0.3085 - val_accuracy: 0.8899
Epoch 5/5
1875/1875 [==============================] - 6s 3ms/step - loss: 0.2332 - accuracy: 0.9154 - val_loss: 0.2858 - val_accuracy: 0.8988
Test Accuracy: 89.880
Training Time (GPU): 35.37 seconds
```

Figure 2: The output of the training on GPU

Typically, the training on a GPU should be significantly faster compared to the training on a CPU. Our results align with this.

c) *Exercise 2 : Take a look at the yellow part of the code above*

In the first exercise, we are asked to train the model several times, but with a different number of filters each time.

Number of filters	8	16	32	64	128
System accuracy	89.120	88.740	89.080	89.970	88.040

Table 1 : System performance evaluation for various numbers of filters in the convolutional layer

```
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.7462 - loss: 0.6961 - val_accuracy: 0.8632 - val_loss: 0.3852
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - accuracy: 0.8764 - loss: 0.3441 - val_accuracy: 0.8791 - val_loss: 0.3372
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - accuracy: 0.8919 - loss: 0.2995 - val_accuracy: 0.8819 - val_loss: 0.3314
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 1ms/step - accuracy: 0.8990 - loss: 0.2747 - val_accuracy: 0.8722 - val_loss: 0.3396
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9065 - loss: 0.2557 - val_accuracy: 0.8879 - val_loss: 0.3093
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9080 - loss: 0.2465 - val_accuracy: 0.8783 - val_loss: 0.3280
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9129 - loss: 0.2382 - val_accuracy: 0.8911 - val_loss: 0.3105
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 3s 2ms/step - accuracy: 0.9153 - loss: 0.2278 - val_accuracy: 0.8912 - val_loss: 0.3037
Number of Filters: 8
Test Accuracy: 89.120
Convergence Time: 30.90 seconds
```

```
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.7264 - loss: 0.7244 - val_accuracy: 0.8574 - val_loss: 0.3928
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 8s 4ms/step - accuracy: 0.8761 - loss: 0.3478 - val_accuracy: 0.8794 - val_loss: 0.3392
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 11s 6ms/step - accuracy: 0.8906 - loss: 0.3060 - val_accuracy: 0.8867 - val_loss: 0.3126
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 11s 6ms/step - accuracy: 0.8996 - loss: 0.2730 - val_accuracy: 0.8874 - val_loss: 0.3119
Number of Filters: 16
Test Accuracy: 88.740
Convergence Time: 42.05 seconds
```

```
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 3ms/step - accuracy: 0.5294 - loss: 1.2052 - val_accuracy: 0.8347 - val_loss: 0.4565
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 11s 6ms/step - accuracy: 0.8550 - loss: 0.4017 - val_accuracy: 0.8638 - val_loss: 0.3769
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 13s 7ms/step - accuracy: 0.8840 - loss: 0.3227 - val_accuracy: 0.8584 - val_loss: 0.3803
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 14s 7ms/step - accuracy: 0.8967 - loss: 0.2868 - val_accuracy: 0.8790 - val_loss: 0.3385
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 11s 6ms/step - accuracy: 0.9049 - loss: 0.2614 - val_accuracy: 0.8769 - val_loss: 0.3346
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9092 - loss: 0.2463 - val_accuracy: 0.8899 - val_loss: 0.3286
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9174 - loss: 0.2259 - val_accuracy: 0.8908 - val_loss: 0.3273
Number of Filters: 32
Test Accuracy: 89.080
Convergence Time: 75.04 seconds
```

```
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 14s 7ms/step - accuracy: 0.7128 - loss: 0.7978 - val_accuracy: 0.8688 - val_loss: 0.3717
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 15s 8ms/step - accuracy: 0.8762 - loss: 0.3412 - val_accuracy: 0.8710 - val_loss: 0.3679
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 10s 5ms/step - accuracy: 0.8974 - loss: 0.2857 - val_accuracy: 0.8827 - val_loss: 0.3290
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 9s 5ms/step - accuracy: 0.9076 - loss: 0.2551 - val_accuracy: 0.8774 - val_loss: 0.3503
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 9s 5ms/step - accuracy: 0.9117 - loss: 0.2414 - val_accuracy: 0.8963 - val_loss: 0.2942
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 9s 5ms/step - accuracy: 0.9178 - loss: 0.2230 - val_accuracy: 0.8950 - val_loss: 0.3126
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 9s 5ms/step - accuracy: 0.9244 - loss: 0.2083 - val_accuracy: 0.8891 - val_loss: 0.3145
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 9s 5ms/step - accuracy: 0.9289 - loss: 0.1914 - val_accuracy: 0.8998 - val_loss: 0.2975
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 9s 5ms/step - accuracy: 0.9324 - loss: 0.1823 - val_accuracy: 0.8997 - val_loss: 0.3053
Number of Filters: 64
Test Accuracy: 89.970
Convergence Time: 103.95 seconds
```

```
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 21s 11ms/step - accuracy: 0.2172 - loss: 1.8731 - val_accuracy: 0.2818 - val_loss: 1.6216
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 21s 11ms/step - accuracy: 0.4957 - loss: 1.2305 - val_accuracy: 0.8267 - val_loss: 0.5009
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 21s 11ms/step - accuracy: 0.8525 - loss: 0.4336 - val_accuracy: 0.8623 - val_loss: 0.3770
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 21s 11ms/step - accuracy: 0.8896 - loss: 0.3142 - val_accuracy: 0.8879 - val_loss: 0.3214
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 21s 11ms/step - accuracy: 0.9031 - loss: 0.2722 - val_accuracy: 0.8760 - val_loss: 0.3542
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 21s 11ms/step - accuracy: 0.9127 - loss: 0.2424 - val_accuracy: 0.8701 - val_loss: 0.3660
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 24s 13ms/step - accuracy: 0.9192 - loss: 0.2259 - val_accuracy: 0.8880 - val_loss: 0.3322
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 24s 13ms/step - accuracy: 0.9263 - loss: 0.2058 - val_accuracy: 0.8970 - val_loss: 0.3101
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 22s 12ms/step - accuracy: 0.9308 - loss: 0.1932 - val_accuracy: 0.8942 - val_loss: 0.3197
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 22s 12ms/step - accuracy: 0.9371 - loss: 0.1800 - val_accuracy: 0.8804 - val_loss: 0.3667
Number of Filters: 128
Test Accuracy: 88.040
Convergence Time: 231.72 seconds
```


The test accuracy varies slightly with the number of filters in the convolutional layer. There isn't a clear pattern indicating that increasing or decreasing the number of filters consistently improves or worsens the accuracy. However, the accuracy remains relatively high across different filter numbers, with the highest accuracy observed when using 64 filters.

The convergence time generally increases as the number of filters in the convolutional layer increases. This could be expected because more filters lead to a larger number of parameters in the model, resulting in longer training times. The convergence time increases notably when using 128 filters, indicating that a higher number of filters significantly increases the training time.

d) *Exercise 3 : Take a look at the pink part of the code above*

In this part, we will modify the number of neurons in the dense hidden layer as specified below in table 2. Meanwhile, we will assign 32 as the number of filters in the convolutional layer.

Number of neurons	16	64	128	256	512

System accuracy	89.790	90.240	91.250	90.080	90.630

Table 2 : System performance evaluation for various numbers of neurons in the dense hidden layer

```
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.7675 - loss: 0.6434 - val_accuracy: 0.8724 - val_loss: 0.3526
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.8878 - loss: 0.3151 - val_accuracy: 0.8861 - val_loss: 0.3173
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9043 - loss: 0.2661 - val_accuracy: 0.8979 - val_loss: 0.2890
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9133 - loss: 0.2393 - val_accuracy: 0.8979 - val_loss: 0.2918
Number of neurons: 16
Test Accuracy: 89.790
Convergence Time: 25.86 seconds
```

```
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 3ms/step - accuracy: 0.7865 - loss: 0.5928 - val_accuracy: 0.8676 - val_loss: 0.3617
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8916 - loss: 0.3011 - val_accuracy: 0.8900 - val_loss: 0.2954
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9061 - loss: 0.2537 - val_accuracy: 0.8942 - val_loss: 0.2940
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9183 - loss: 0.2217 - val_accuracy: 0.8961 - val_loss: 0.2866
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9275 - loss: 0.1998 - val_accuracy: 0.9020 - val_loss: 0.2778
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9329 - loss: 0.1831 - val_accuracy: 0.9024 - val_loss: 0.2788
Number of neurons: 64
Test Accuracy: 90.240
Convergence Time: 44.10 seconds
```

```
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 11s 5ms/step - accuracy: 0.7929 - loss: 0.5652 - val_accuracy: 0.8659 - val_loss: 0.370
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 10s 6ms/step - accuracy: 0.8983 - loss: 0.2798 - val_accuracy: 0.8964 - val_loss: 0.278
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 11s 6ms/step - accuracy: 0.9123 - loss: 0.2382 - val_accuracy: 0.8981 - val_loss: 0.284
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 9s 5ms/step - accuracy: 0.9231 - loss: 0.2095 - val_accuracy: 0.9021 - val_loss: 0.2721
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 9s 5ms/step - accuracy: 0.9343 - loss: 0.1829 - val_accuracy: 0.8969 - val_loss: 0.2924
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 14s 8ms/step - accuracy: 0.9370 - loss: 0.1653 - val_accuracy: 0.9054 - val_loss: 0.280
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 16s 9ms/step - accuracy: 0.9447 - loss: 0.1483 - val_accuracy: 0.9144 - val_loss: 0.263
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 16s 9ms/step - accuracy: 0.9518 - loss: 0.1301 - val_accuracy: 0.9106 - val_loss: 0.284
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 16s 8ms/step - accuracy: 0.9582 - loss: 0.1148 - val_accuracy: 0.9129 - val_loss: 0.270
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 16s 8ms/step - accuracy: 0.9646 - loss: 0.1012 - val_accuracy: 0.9125 - val_loss: 0.278
Number of neurons: 128
Test Accuracy: 91.250
Convergence Time: 142.70 seconds
```

```
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 23s 12ms/step - accuracy: 0.7956 - loss: 0.5656 - val_accuracy: 0.8668 - val_loss: 0.3581
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 23s 12ms/step - accuracy: 0.8944 - loss: 0.2926 - val_accuracy: 0.8857 - val_loss: 0.3105
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 21s 11ms/step - accuracy: 0.9127 - loss: 0.2360 - val_accuracy: 0.9016 - val_loss: 0.2673
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 23s 12ms/step - accuracy: 0.9257 - loss: 0.2032 - val_accuracy: 0.9008 - val_loss: 0.2752
Number of neurons: 256
Test Accuracy: 90.080
Convergence Time: 98.54 seconds
```

```
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 31s 16ms/step - accuracy: 0.8042 - loss: 0.5411 - val_accuracy: 0.8888 - val_loss: 0.3081
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 39s 21ms/step - accuracy: 0.8983 - loss: 0.2757 - val_accuracy: 0.8966 - val_loss: 0.2869
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 36s 19ms/step - accuracy: 0.9181 - loss: 0.2221 - val_accuracy: 0.9026 - val_loss: 0.2657
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 36s 19ms/step - accuracy: 0.9310 - loss: 0.1921 - val_accuracy: 0.9063 - val_loss: 0.2717
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 37s 20ms/step - accuracy: 0.9398 - loss: 0.1638 - val_accuracy: 0.9063 - val_loss: 0.2589
Number of neurons: 512
Test Accuracy: 90.630
Convergence Time: 189.68 seconds
```

The test accuracy tends to increase as the number of neurons in the model increases, indicating that a larger number of neurons allows the model to capture more complex patterns in the data, leading to improved performance on unseen data. However, this relationship is not necessarily linear.

In the case where 128 neurons are used, while the accuracy is higher, the convergence time is significantly longer. Therefore, it becomes essential to strike a balance between convergence time (to decrease computations) and test accuracy (to increase accuracy).

e) Exercise 4 : Take a look at the purple part of the code above

In this exercise, we will modify the number of epochs used to train the model as specified below in Table 3. We will use 16 neurons in the dense hidden layer and 32 filters in the convolutional layer.

Number of epochs	1	2	5	10	20

System accuracy	87.100	87.110	88.340	89.860	89.730

Table 3. System performance evaluation for different values of the number of epochs

```
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.7640 - loss: 0.6420 - val_accuracy: 0.8710 - val_loss: 0.3697
Train the model with : 1 epochs
Test Accuracy: 87.100

Epoch 1/2
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 3ms/step - accuracy: 0.7625 - loss: 0.6484 - val_accuracy: 0.8752 - val_loss: 0.3598
Epoch 2/2
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.8802 - loss: 0.3395 - val_accuracy: 0.8711 - val_loss: 0.3544
Train the model with : 2 epochs
Test Accuracy: 87.110

Epoch 1/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.6859 - loss: 0.8168 - val_accuracy: 0.8453 - val_loss: 0.4611
Epoch 2/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.8695 - loss: 0.3730 - val_accuracy: 0.8745 - val_loss: 0.3622
Epoch 3/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.8928 - loss: 0.3043 - val_accuracy: 0.8841 - val_loss: 0.3265
Epoch 4/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9048 - loss: 0.2713 - val_accuracy: 0.8900 - val_loss: 0.3118
Epoch 5/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9109 - loss: 0.2480 - val_accuracy: 0.8834 - val_loss: 0.3440
Train the model with : 5 epochs
Test Accuracy: 88.340

Epoch 1/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.7178 - loss: 0.7829 - val_accuracy: 0.8620 - val_loss: 0.4074
Epoch 2/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.8735 - loss: 0.3599 - val_accuracy: 0.8710 - val_loss: 0.3637
Epoch 3/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 9s 5ms/step - accuracy: 0.8912 - loss: 0.3095 - val_accuracy: 0.8794 - val_loss: 0.3456
Epoch 4/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 8s 4ms/step - accuracy: 0.8985 - loss: 0.2845 - val_accuracy: 0.8873 - val_loss: 0.3170
Epoch 5/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9049 - loss: 0.2607 - val_accuracy: 0.8842 - val_loss: 0.3458
Epoch 6/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.9125 - loss: 0.2448 - val_accuracy: 0.8919 - val_loss: 0.3097
Epoch 7/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.9174 - loss: 0.2283 - val_accuracy: 0.8910 - val_loss: 0.3234
Epoch 8/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 10s 4ms/step - accuracy: 0.9178 - loss: 0.2225 - val_accuracy: 0.8904 - val_loss: 0.3209
Epoch 9/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9241 - loss: 0.2093 - val_accuracy: 0.8803 - val_loss: 0.3515
Epoch 10/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 8s 4ms/step - accuracy: 0.9243 - loss: 0.2091 - val_accuracy: 0.8986 - val_loss: 0.3140
Train the model with : 10 epochs
Test Accuracy: 89.860

Epoch 4/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.9034 - loss: 0.2713 - val_accuracy: 0.8818 - val_loss: 0.3196
Epoch 5/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.9090 - loss: 0.2538 - val_accuracy: 0.8895 - val_loss: 0.3058
Epoch 6/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 9s 5ms/step - accuracy: 0.9155 - loss: 0.2322 - val_accuracy: 0.8914 - val_loss: 0.3053
Epoch 7/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.9197 - loss: 0.2170 - val_accuracy: 0.8912 - val_loss: 0.3095
Epoch 8/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9227 - loss: 0.2104 - val_accuracy: 0.8984 - val_loss: 0.3004
Epoch 9/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9272 - loss: 0.1997 - val_accuracy: 0.8979 - val_loss: 0.2993
Epoch 10/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 8s 4ms/step - accuracy: 0.9323 - loss: 0.1830 - val_accuracy: 0.8991 - val_loss: 0.2921
Epoch 11/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 8s 4ms/step - accuracy: 0.9372 - loss: 0.1718 - val_accuracy: 0.9013 - val_loss: 0.2882
Epoch 12/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9377 - loss: 0.1693 - val_accuracy: 0.8966 - val_loss: 0.3132
Epoch 13/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9400 - loss: 0.1594 - val_accuracy: 0.9022 - val_loss: 0.2869
Epoch 14/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9446 - loss: 0.1551 - val_accuracy: 0.8990 - val_loss: 0.3127
Epoch 15/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.9456 - loss: 0.1506 - val_accuracy: 0.8981 - val_loss: 0.3132
Epoch 16/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9497 - loss: 0.1374 - val_accuracy: 0.9036 - val_loss: 0.3058
Epoch 17/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9509 - loss: 0.1355 - val_accuracy: 0.8969 - val_loss: 0.3329
Epoch 18/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.9528 - loss: 0.1308 - val_accuracy: 0.9054 - val_loss: 0.3214
Epoch 19/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.9523 - loss: 0.1296 - val_accuracy: 0.8999 - val_loss: 0.3454
Epoch 20/20
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9556 - loss: 0.1195 - val_accuracy: 0.8973 - val_loss: 0.3419
Train the model with : 20 epochs
Test Accuracy: 89.730
```

We can notice that as the number of epochs increases, the test accuracy generally improves. However, there seems to be some fluctuations in terms of accuracy with the increase in the number of epochs, which could be due to various factors such as overfitting or the model reaching a plateau in learning.

f) ***Exercise 5 : Take a look at the blue part of the code above***

In this exercise, we will modify the learning rate of the SGD optimizer. So, we will maintain the epochs fixed to 5 to train the model.

Learning rate	0.1	0.01	0.001	0.0001	0.00001
System accuracy	84.970	88.650	87.470	83.930	69.550

Table 4. : System performance evaluation for various learning rates of the SGD

```
Epoch 1/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.7443 - loss: 0.7413 - val_accuracy: 0.8216 - val_loss: 0.5189
Epoch 2/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.8242 - loss: 0.5009 - val_accuracy: 0.8383 - val_loss: 0.4763
Epoch 3/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.8448 - loss: 0.4440 - val_accuracy: 0.8037 - val_loss: 0.5509
Epoch 4/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.8498 - loss: 0.4274 - val_accuracy: 0.8386 - val_loss: 0.5471
Epoch 5/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 4s 2ms/step - accuracy: 0.8589 - loss: 0.4152 - val_accuracy: 0.8497 - val_loss: 0.4694
Train the model with : 0.1 as a learning rate
Test Accuracy: 84.970
```

```
Epoch 1/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.6566 - loss: 0.8904 - val_accuracy: 0.8411 - val_loss: 0.4658
Epoch 2/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.8684 - loss: 0.3689 - val_accuracy: 0.8770 - val_loss: 0.3673
Epoch 3/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.8880 - loss: 0.3225 - val_accuracy: 0.8843 - val_loss: 0.3379
Epoch 4/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 2ms/step - accuracy: 0.8996 - loss: 0.2866 - val_accuracy: 0.8862 - val_loss: 0.3244
Epoch 5/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.9080 - loss: 0.2623 - val_accuracy: 0.8865 - val_loss: 0.3125
Train the model with : 0.01 as a learning rate
Test Accuracy: 88.650
```

```
Epoch 1/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.7085 - loss: 0.8609 - val_accuracy: 0.8238 - val_loss: 0.4805
Epoch 2/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 8s 4ms/step - accuracy: 0.8541 - loss: 0.4187 - val_accuracy: 0.8623 - val_loss: 0.3927
Epoch 3/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.8716 - loss: 0.3642 - val_accuracy: 0.8617 - val_loss: 0.3887
Epoch 4/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8805 - loss: 0.3377 - val_accuracy: 0.8778 - val_loss: 0.3471
Epoch 5/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8892 - loss: 0.3189 - val_accuracy: 0.8747 - val_loss: 0.3521
Train the model with : 0.001 as a learning rate
Test Accuracy: 87.470
```

```
Epoch 1/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 8s 4ms/step - accuracy: 0.5182 - loss: 1.4626 - val_accuracy: 0.7773 - val_loss: 0.6709
Epoch 2/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.7948 - loss: 0.6205 - val_accuracy: 0.8091 - val_loss: 0.5588
Epoch 3/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 3ms/step - accuracy: 0.8220 - loss: 0.5277 - val_accuracy: 0.8254 - val_loss: 0.5100
Epoch 4/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8345 - loss: 0.4841 - val_accuracy: 0.8303 - val_loss: 0.4917
Epoch 5/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 5s 3ms/step - accuracy: 0.8432 - loss: 0.4589 - val_accuracy: 0.8393 - val_loss: 0.4655
Train the model with : 0.0001 as a learning rate
Test Accuracy: 83.930
```

```
Epoch 1/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.3047 - loss: 2.0237 - val_accuracy: 0.5527 - val_loss: 1.4711
Epoch 2/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 8s 4ms/step - accuracy: 0.5695 - loss: 1.4043 - val_accuracy: 0.6160 - val_loss: 1.2651
Epoch 3/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 4ms/step - accuracy: 0.6319 - loss: 1.2177 - val_accuracy: 0.6472 - val_loss: 1.1088
Epoch 4/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.6598 - loss: 1.0702 - val_accuracy: 0.6781 - val_loss: 0.9850
Epoch 5/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.6956 - loss: 0.9482 - val_accuracy: 0.6955 - val_loss: 0.9083
Train the model with : 1e-05 as a learning rate
Test Accuracy: 69.550
```

From these results, it seems that a learning rate of 0.01 achieved the highest test accuracy of 88.650%. Lower learning rates (0.001, 0.0001, 0.00001) and higher learning rates (0.1) yielded lower accuracies.

These results suggest that choosing an appropriate learning rate is crucial for achieving optimal performance in training neural networks. A learning rate that is too high can cause the model to diverge, while a learning rate that is too low can lead to slow convergence or getting stuck in local minima.

g) *Exercise 6 : Take a look at the green part of the code above*

In this part, we will change the dropout percentage, which means we will deactivate a certain number of neurons randomly in order to prevent overfitting by randomly dropping a fraction of input units during training.

Dropout Percentage	0.1	0.2	0.3	0.4	0.5
System accuracy	88.340	89.600	89.340	88.580	88.510

Table 5. System performance evaluation for number of neurons dropped in the dropout layer

```
Epoch 1/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 3ms/step - accuracy: 0.7171 - loss: 0.7545 - val_accuracy: 0.8466 - val_loss: 0.4350
Epoch 2/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8662 - loss: 0.3827 - val_accuracy: 0.8590 - val_loss: 0.4134
Epoch 3/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8845 - loss: 0.3304 - val_accuracy: 0.8713 - val_loss: 0.3677
Epoch 4/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8930 - loss: 0.3026 - val_accuracy: 0.8881 - val_loss: 0.3175
Epoch 5/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9020 - loss: 0.2747 - val_accuracy: 0.8834 - val_loss: 0.3294
Train the model with : 0.1 as a deopout percentage
Test Accuracy: 88.340
Epoch 1/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.7372 - loss: 0.7110 - val_accuracy: 0.8641 - val_loss: 0.3758
Epoch 2/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8740 - loss: 0.3541 - val_accuracy: 0.8726 - val_loss: 0.3432
Epoch 3/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8879 - loss: 0.3107 - val_accuracy: 0.8770 - val_loss: 0.3500
Epoch 4/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8985 - loss: 0.2839 - val_accuracy: 0.8921 - val_loss: 0.2983
Epoch 5/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.9045 - loss: 0.2654 - val_accuracy: 0.8960 - val_loss: 0.2882
Train the model with : 0.2 as a deopout percentage
Test Accuracy: 89.600
```

```
Epoch 1/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 3ms/step - accuracy: 0.7170 - loss: 0.7474 - val_accuracy: 0.8544 - val_loss: 0.4192
Epoch 2/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8672 - loss: 0.3742 - val_accuracy: 0.8669 - val_loss: 0.3713
Epoch 3/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8835 - loss: 0.3258 - val_accuracy: 0.8842 - val_loss: 0.3273
Epoch 4/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8915 - loss: 0.3044 - val_accuracy: 0.8697 - val_loss: 0.3591
Epoch 5/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8971 - loss: 0.2869 - val_accuracy: 0.8934 - val_loss: 0.3066
Train the model with : 0.3 as a deopout percentage
Test Accuracy: 89.340
```

```
Epoch 1/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 3ms/step - accuracy: 0.5139 - loss: 1.2461 - val_accuracy: 0.8114 - val_loss: 0.5106
Epoch 2/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8234 - loss: 0.4970 - val_accuracy: 0.8533 - val_loss: 0.4012
Epoch 3/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8619 - loss: 0.3939 - val_accuracy: 0.8744 - val_loss: 0.3543
Epoch 4/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8768 - loss: 0.3483 - val_accuracy: 0.8790 - val_loss: 0.3324
Epoch 5/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8829 - loss: 0.3224 - val_accuracy: 0.8855 - val_loss: 0.3186
Train the model with : 0.4 as a deopout percentage
Test Accuracy: 88.550
```

```
Epoch 1/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 7s 3ms/step - accuracy: 0.7689 - loss: 0.6349 - val_accuracy: 0.8647 - val_loss: 0.3813
Epoch 2/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8633 - loss: 0.3709 - val_accuracy: 0.8785 - val_loss: 0.3438
Epoch 3/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8757 - loss: 0.3395 - val_accuracy: 0.8769 - val_loss: 0.3335
Epoch 4/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8841 - loss: 0.3151 - val_accuracy: 0.8919 - val_loss: 0.3035
Epoch 5/5
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 6s 3ms/step - accuracy: 0.8878 - loss: 0.3034 - val_accuracy: 0.8851 - val_loss: 0.3097
Train the model with : 0.5 as a deopout percentage
Test Accuracy: 88.510
```

Based on these results, it seems that a dropout percentage of 0.2 achieved the highest test accuracy of 89.6%. This suggests that adding dropout with a rate of 0.2 during training helped improve generalization and reduce overfitting in your model. Meanwhile, the results confirm that a dropout rate of 0.4 or 0.5 might be too aggressive for this model, leading to suboptimal performance.

h) *Exercise 7 : we have just used the optimal values in the code*

In this exercise we will summarize the given results obtained above by extracting the optimal values for all used parameters. The table below specifies the best values saved :

Parameter	Number of filters	Number of neurons	Number of epochs	Learning rate	Dropout percentage
Optimal values	64	128	10	0.01	0.2

Table 6 : Optimal values for used parameters

When we used all these values to train the model we obtained a test accuracy as above that reached 91.140 which is a good accuracy.

```
Epoch 1/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 22s 11ms/step - accuracy: 0.7870 - loss: 0.5867 - val_accuracy: 0.8694 - val_loss: 0.3552
Epoch 2/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 21s 11ms/step - accuracy: 0.8924 - loss: 0.2932 - val_accuracy: 0.8991 - val_loss: 0.2871
Epoch 3/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 21s 11ms/step - accuracy: 0.9081 - loss: 0.2451 - val_accuracy: 0.9003 - val_loss: 0.2746
Epoch 4/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 21s 11ms/step - accuracy: 0.9189 - loss: 0.2218 - val_accuracy: 0.8962 - val_loss: 0.2905
Epoch 5/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 21s 11ms/step - accuracy: 0.9257 - loss: 0.1973 - val_accuracy: 0.9036 - val_loss: 0.2622
Epoch 6/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 21s 11ms/step - accuracy: 0.9341 - loss: 0.1826 - val_accuracy: 0.9076 - val_loss: 0.2671
Epoch 7/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 22s 12ms/step - accuracy: 0.9398 - loss: 0.1637 - val_accuracy: 0.9111 - val_loss: 0.2551
Epoch 8/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 22s 12ms/step - accuracy: 0.9439 - loss: 0.1529 - val_accuracy: 0.9030 - val_loss: 0.2773
Epoch 9/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 22s 12ms/step - accuracy: 0.9485 - loss: 0.1399 - val_accuracy: 0.9072 - val_loss: 0.2742
Epoch 10/10
1875/1875 ━━━━━━━━━━━━━━━━━━━━ 22s 11ms/step - accuracy: 0.9514 - loss: 0.1298 - val_accuracy: 0.9114 - val_loss: 0.2739
Test Accuracy: 91.140
```


i) Exercise 8 : Take a look at the orange part of the code above

II) Application II :

This application used a different training strategy using three datasets : one for training, one for validation and the final one for testing purposes. Also, this strategy used k-folds cross-validation algorithm to create a validation dataset. It splits the training dataset into K folds and trains K models, each using different folds for training and validation.
This method can be considered efficient due to a better estimation of model performance because it averages the performance across multiple folds.

Below, you will find the code of all steps and exercises :

a) Code of all the application :

```
# Application 2 - we used another file to implement the code just to make things clearer
import numpy as np
from sklearn.model_selection import KFold
import keras
from keras.optimizers import SGD
from keras.datasets import fashion_mnist
from keras.utils import to_categorical
from keras import layers
from keras.layers import Dropout
from matplotlib import pyplot
from keras.models import load_model
import cv2
import time

# Define the function to summarize learning curves and performances
def summarizeLearningCurvesPerformances(histories, accuracyScores):
    for i in range(len(histories)):
        # plot loss
        pyplot.subplot(211)
        pyplot.title('Cross Entropy Loss')
```

```
        pyplot.plot(histories[i].history['loss'], color='green', label='train')
        pyplot.plot(histories[i].history['val_loss'], color='red', label='test')

        # plot accuracy
        pyplot.subplot(212)
        pyplot.title('Classification Accuracy')
        pyplot.plot(histories[i].history['accuracy'], color='green', label='train')
        pyplot.plot(histories[i].history['val_accuracy'], color='red', label='test')

        # Print accuracy for each split
        print("Accuracy for set {} = {}".format(i, accuracyScores[i]))

    pyplot.show()

    print('Accuracy: mean = {:.3f} std = {:.3f}, n = {}'.format(np.mean(accuracyScores) * 100,
np.std(accuracyScores) * 100, len(accuracyScores)))

# Define the function to prepare the data
def prepareData(trainX, trainY, testX, testY):
    trainX = trainX.reshape((trainX.shape[0], 28, 28, 1))
    testX = testX.reshape((testX.shape[0], 28, 28, 1))
    trainX = trainX.astype('float32') / 255
    testX = testX.astype('float32') / 255
    trainY = to_categorical(trainY)
    testY = to_categorical(testY)
    return trainX, trainY, testX, testY

#we used the same CNN architecture as the application 1
def defineModel(input_shape, num_classes):
    model = keras.Sequential()
    #we added a parameter padding = same and increasing the number of filters in the line below
as indicated in exercise9 application2
    model.add(layers.Conv2D(64, (3, 3),padding='same', activation='relu',
kernel_initializer='he_uniform', input_shape=input_shape))
    model.add(layers.MaxPooling2D((2, 2)))
    model.add(Dropout(0.2))
    model.add(layers.Flatten())
    model.add(layers.Dense(128, activation='relu', kernel_initializer='he_uniform'))
    model.add(layers.Dense(num_classes, activation='softmax'))
    model.compile(optimizer=SGD(learning_rate=0.01, momentum=0.9),
loss='categorical_crossentropy', metrics=['accuracy'])
    return model

# Define the function to train and evaluate using k-fold cross-validation
def defineTrainAndEvaluateKFolds(trainX, trainY, testX, testY):
    #Application 2 - Step 2 - Prepare the cross validation datasets
    k_folds = 5
    accuracyScores = []
    histories = []
    kfold = KFold(k_folds, shuffle=True, random_state=1)
    for train_idx, val_idx in kfold.split(trainX):
        #TODO - Application 2 - Step 3 - Select data for train and validation
```

```
        trainX_i, trainY_i = trainX[train_idx], trainY[train_idx]
        valX_i, valY_i = trainX[val_idx], trainY[val_idx]
        #TODO - Application 2 - Step 4 - Build the model - Call the defineModel function
        model = defineModel((28, 28, 1), 10)
        #TODO - Application 2 - Step 5 - Fit the model
        history = model.fit(trainX_i, trainY_i, epochs=5, batch_size=32, validation_data=(valX_i,
valY_i), verbose=1)
        #TODO - Application 2 - Step 6 - Save the training related information in the histories
list
        histories.append(history)
        #TODO - Application 2 - Step 7 - Evaluate the model on the test dataset
        loss, accuracy = model.evaluate(testX, testY, verbose=0)
        #TODO - Application 2 - Step 8 - Save the accuracy in the accuracyScores list
        accuracyScores.append(accuracy)
    return histories, accuracyScores


def main():
    #we used the same code as the application 1 for the data preparation
    (trainX, trainY), (testX, testY) = fashion_mnist.load_data()
    print('Train:', trainX.shape, trainY.shape)
    print('Test:', testX.shape, testY.shape)
    trainX, trainY, testX, testY = prepareData(trainX, trainY, testX, testY)

    #TODO - Application 2 - Step 1 - Define, train and evaluate the model using K-Folds strategy
    histories, accuracyScores = defineTrainAndEvaluateKFolds(trainX, trainY, testX, testY)

    #TODO - Application 2 - Step9 - System performance presentation
    summarizeLearningCurvesPerformances(histories, accuracyScores)

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

b) The output of all steps :

```
Epoch 1/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 19s 12ms/step - accuracy: 0.7754 - loss: 0.6194 - val_accuracy: 0.8875 - val_loss:
0.3065
Epoch 2/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 17s 12ms/step - accuracy: 0.8918 - loss: 0.2975 - val_accuracy: 0.8983 - val_loss:
0.2756
Epoch 3/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.9073 - loss: 0.2535 - val_accuracy: 0.9054 - val_loss:
```

```
0.2609
Epoch 4/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.9146 - loss: 0.2266 - val_accuracy: 0.9089 - val_loss: 0.2533
Epoch 5/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 17s 11ms/step - accuracy: 0.9251 - loss: 0.2004 - val_accuracy: 0.9032 - val_loss: 0.2655
Epoch 1/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 20s 13ms/step - accuracy: 0.7933 - loss: 0.5831 - val_accuracy: 0.8862 - val_loss: 0.3128
Epoch 2/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.8917 - loss: 0.2971 - val_accuracy: 0.9012 - val_loss: 0.2711
Epoch 3/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 19s 13ms/step - accuracy: 0.9088 - loss: 0.2492 - val_accuracy: 0.9035 - val_loss: 0.2622
Epoch 4/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 20s 13ms/step - accuracy: 0.9205 - loss: 0.2154 - val_accuracy: 0.9118 - val_loss: 0.2413
Epoch 5/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 17s 11ms/step - accuracy: 0.9267 - loss: 0.1961 - val_accuracy: 0.9073 - val_loss: 0.2616
Epoch 1/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.7748 - loss: 0.6197 - val_accuracy: 0.8853 - val_loss: 0.3369
Epoch 2/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 17s 11ms/step - accuracy: 0.8887 - loss: 0.3005 - val_accuracy: 0.8923 - val_loss: 0.3151
Epoch 3/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 17s 12ms/step - accuracy: 0.9074 - loss: 0.2556 - val_accuracy: 0.9050 - val_loss: 0.2749
Epoch 4/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 19s 12ms/step - accuracy: 0.9156 - loss: 0.2283 - val_accuracy: 0.9018 - val_loss: 0.2794
Epoch 5/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.9239 - loss: 0.2049 - val_accuracy: 0.9061 - val_loss: 0.2701
Epoch 1/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 19s 12ms/step - accuracy: 0.7911 - loss: 0.5839 - val_accuracy: 0.8823 - val_loss: 0.3190
Epoch 2/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 17s 11ms/step - accuracy: 0.8934 - loss: 0.2986 - val_accuracy: 0.8925 - val_loss: 0.2868
Epoch 3/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 15s 10ms/step - accuracy: 0.9092 - loss: 0.2479 - val_accuracy: 0.9003 - val_loss: 0.2673
Epoch 4/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 16s 10ms/step - accuracy: 0.9178 - loss: 0.2250 - val_accuracy: 0.9053 - val_loss: 0.2588
Epoch 5/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.9240 - loss: 0.2037 - val_accuracy: 0.9100 - val_loss:
```

```
0.2475
Epoch 1/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 18s 12ms/step - accuracy: 0.7790 - loss: 0.6214 - val_accuracy: 0.8857 - val_loss:
0.3245
Epoch 2/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 16s 11ms/step - accuracy: 0.8879 - loss: 0.3088 - val_accuracy: 0.8958 - val_loss:
0.2907
Epoch 3/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 16s 11ms/step - accuracy: 0.9037 - loss: 0.2589 - val_accuracy: 0.9008 - val_loss:
0.2713
Epoch 4/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 16s 11ms/step - accuracy: 0.9138 - loss: 0.2287 - val_accuracy: 0.9010 - val_loss:
0.2766
Epoch 5/5
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 16s 11ms/step - accuracy: 0.9248 - loss: 0.2037 - val_accuracy: 0.9120 - val_loss:
0.2610
Accuracy for set 0 = 0.8985000252723694
Accuracy for set 1 = 0.8956000208854675
Accuracy for set 2 = 0.8998000025749207
Accuracy for set 3 = 0.9036999940872192
Accuracy for set 4 = 0.9003999829292297
```

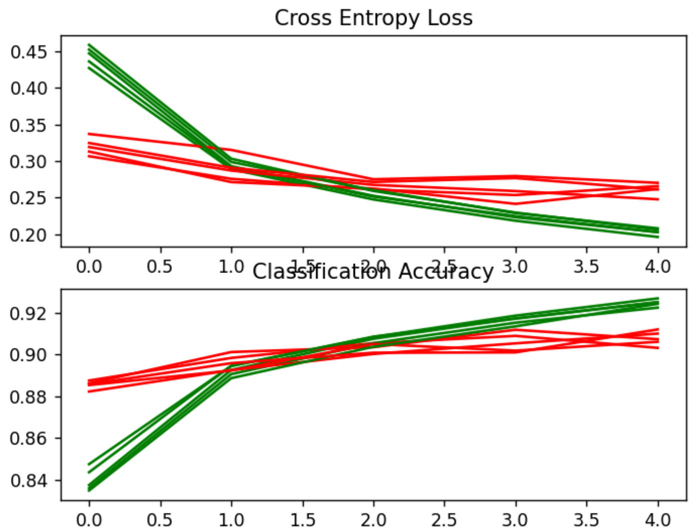

Figure 1 : The output of the training process with k-folds

In the results we obtained, we performed a 5-fold cross-validation (k-folds) to evaluate the model. For each fold, the model was trained on a subset of the data and evaluated on another subset of the data. For each fold, the model was trained for 5 epochs, as specified in the code.
The average performance (accuracy) reported is the average of the performance obtained over the 5 cross-validation folds. This means that we evaluated the performance of the model on 5 different datasets and reported the average of these performances to obtain a more reliable estimate of the overall performance of the model.

Assuming that the green curve represents the training performance and the red one shows the validation performances, we can notice that :

- the average accuracy for each cross-validation fold is between approximately 89.85% and 90.37%, which is relatively high. This suggests that the model is capable of generalizing well on data that it did not see during training.

- The average loss on the cross-validation set decreases over epochs, which is expected when training a model.

c) Exercise 8 :

Application 1 uses a default padding, this leads to a loss of information at the borders : so without padding, the convolution operation reduces the spatial dimensions of the feature maps. This reduction can lead to information loss, especially at the borders of the input images. By using “same” padding, this loss is mitigated, improving the ability of the model to capture features from the entire input image. So, the system accuracy might improve with 'same' padding due to the preservation of spatial information and the reduction of information loss at the borders of the input images.

d) Exercise 9 :

In the first application, when we increased the number of filters, the accuracy improved from 89.790 to 90.240. In fact, increasing the number of filters allows the model to capture more diverse and complex features from the input images.