PRACTICAL SESSION 2 Supervised DEEP LEARNING

The dataset is a *subset* of CIFAR10 (a very popular dataset in Machine Learning) with only 4 classes: **BIRD**, **CAT**, **FROG**, **HORSE**. We will call our dataset **CIFAR4**. It is composed of 24000 images from 4 types of animals.

Input data are square color images (RGB). The size of a picture is 32x32. So we have an input tensor of size [32,32,3]. Each pixel has a red, green and blue UINT8 values, i.e. in [0,255].

The dataset is available on **ECAMPUS**

The main objective is to train different deep neural network models to classify an image into one of the 4 classes.

The goal of this practical session is to experiment! Everything you need is easily available in the TENSORFLOW documentation that contains many examples.

BONUS STRIKE

You can make additional experiences. In that case, please add bonus works at the end of your notebook in the **[BONUS]** section.

Bonus works could bring bonus points on the global note for the practical sessions ***if and only if*** all the mandatory jobs have been done correctly.

IMPORT LIBS

We will use TENSORFLOW, the Deep Learning platform from Google.

TensorFlow is easy to understand and the documentation and tutorials are (very) useful when learning Deep Learning.

I already know how to develop DL models and I want to use PyTorch instead. Is it possible?

==> *YES.* PYTORCH (from Meta) is the other big reference for DL libraries. Note that, in that case, *it's your choice* and you must be *self-sufficient* in case of development issues.

```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import urllib
import zipfile
import matplotlib.pyplot as plt
from PIL import Image
import tensorflow as tf
```

[WARNING] About the use of COLAB and GPU

On the top-right of the colab environment, you can chose the execution environment ("Modify the execution type") between CPU and GPU (Nvidia T4). With a free google account, there is no limitation with the CPU-only mode. With the GPU, you have a limitation that is dynamically set by Google regarding the overall load on their infrastructure. Usually, it could be 1 hour per day.

For this practical session, you can keep the CPU mode for the MLP part. For the CNN section, you can swith to the GPU mode if you consider the processing are too slow. But, be careful and not too GPU-enthusiast.

NB: You can check the CPU info with `!cat /proc/cpuinfo`

```
#Get CPU info
!cat /proc/cpuinfo
#Get GPU info
#!nividia-smi
```

```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 85
model name      : Intel(R) Xeon(R) CPU @ 2.00GHz
stepping        : 3
microcode       : 0xffffffff
cpu MHz         : 2000.192
cache size      : 39424 KB
physical id     : 0
```

```
siblings        : 2
core id         : 0
cpu cores       : 1
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ss ht syscall nx
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf mds swapgs taa mmio_stale_data retbleed bhi
bogomips        : 4000.38
clflush size    : 64
cache_alignment : 64
address sizes   : 46 bits physical, 48 bits virtual
power management:

processor       : 1
vendor_id       : GenuineIntel
cpu family      : 6
model           : 85
model name      : Intel(R) Xeon(R) CPU @ 2.00GHz
stepping        : 3
microcode       : 0xffffffff
cpu MHz         : 2000.192
cache size      : 39424 KB
physical id     : 0
siblings        : 2
core id         : 0
cpu cores       : 1
apicid          : 1
initial apicid  : 1
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ss ht syscall nx
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf mds swapgs taa mmio_stale_data retbleed bhi
bogomips        : 4000.38
clflush size    : 64
cache_alignment : 64
address sizes   : 46 bits physical, 48 bits virtual
power management:
```

DOWNLOAD AND CHECK THE DATASET [3 pts]

Data must be located in a ./data directory at the same level as this notebook

```
#IF YOU USE Google COLAB, you can mount your Google Drive:
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
#Create a 'data/' directory, put the numpy files and load. With Colab case, create 'data' here:
X_cifar4=np.load("/content/drive/MyDrive/Colab Notebooks/data/CIFAR4_X.npy")
Y_cifar4=np.load("/content/drive/MyDrive/Colab Notebooks/data/CIFAR4_Y.npy")
print(np.shape(X_cifar4))
print(np.shape(Y_cifar4))
nb_labels=4
```

```
(24000, 32, 32, 3)
(24000,)
```

[QUESTION] Display some images from X_cifar4 with the corresponding label

NB: Since the pictures are very small, use `plt.figure(figsize=(2, 2), dpi=80)` before `plt.imshow()` to display something *"watchable"*.

```
j = 11995  # Starting index
for i in range(j, j + 9):  # Display 9 images sequentially
    plt.figure(figsize=(2, 2), dpi=80)  # Set figure size for each image
    plt.imshow(X_cifar4[i])  # Display the image
    plt.title(f"Label: {Y_cifar4[i]}")  # Add the label as the title
    plt.axis('off')  # Remove axis for clarity
    plt.show()  # Show the image
```

A pixel is an UINT8 value, so in [0;255]. We will normalize data in [0,1]:

```
print("First pixel (r,g,b) of the first image:", X_cifar4[0,0,0,:])
X = X_cifar4/255.0
print("Now in [0,1] ==> ", X[0,0,0,:])
```

```
First pixel (r,g,b) of the first image: [164 206  84]
Now in [0,1] ==>  [0.64313725 0.80784314 0.32941176]
```

For the labels **Y**, usually, we prefer to process "one-hot encodings" i.e., a vector with '0' everywhere except for the corresponding label where you have '1'.

Example : let's say you have 4 labels and 10 training data with the original Y:

```
Y=[0,1,1,3,3,3,2,2,1,0]
shape(Y)=[10,]
```

Then the "one-hot encoding" version of Y will be:

```
Y_onehot=
[1,0,0,0;
 0,1,0,0;
 0,1,0,0;
 0,0,0,1;
 0,0,0,1;
 0,0,0,1;
 0,0,1,0;
 0,0,1,0;
 0,1,0,0;
 1,0,0,0]
shape(Y_onehot)=[10,4]
```

[QUESTION] Use the tensorflow method *tf.keras.utils.to_categorical()* to transform your Y_cifar4 into Y and check the shape of your new Y.

```
import keras
#Your code here

Y = keras.utils.to_categorical(Y_cifar4, num_classes=4)
print(Y)
```

```
[[1. 0. 0. 0.]
 [1. 0. 0. 0.]
 [1. 0. 0. 0.]
 ...
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]
 [0. 0. 0. 1.]]
```

CREATE A TRAINING/VALIDATION/TEST dataset [2 pt]

NB: This step is similar to the supervised section in TP1.

[QUESTION] Use the SKLEARN method train_test_split in order to create:

- a TRAIN set (X_train,Y_train) *[advice: use 70% of the whole dataset]*
- a VALIDATION set (X_val, Y_val) *[advice: 15%]*
- a TEST set (X_test, Y_test) *[advice: 15%]*

Print the shape of the 3 datasets.

The TRAIN and VALIDATION sets will be used at training time. The TEST set will be used, after the training, at the inference time.

NOTE that we could only use a single VALIDATION/TEST set for both the training and the inference stages.

```
#Your code here
#CREATE A TRAINING/ VALIDATION/TEST dataset
from sklearn.model_selection import train_test_split

# Split the dataset into training and test sets
X_train, X_test, Y_train, Y_test= train_test_split(X_cifar4, Y, train_size=0.7, random_state=42)
X_val, X_test, Y_val, Y_test= train_test_split(X_test, Y_test, train_size=0.5, random_state=42)

# Check the sizes of the resulting datasets
print(f"Size of X_train: {X_train.shape}")
print(f"Size of Y_train: {Y_train.shape}")
```

```
print(f"Size of X_test: {X_test.shape}")
print(f"Size of Y_test: {Y_test.shape}")
print(f"Size of X_val: {X_val.shape}")
print(f"Size of Y_val: {Y_val.shape}")
```

```
Size of X_train: (16800, 32, 32, 3)
Size of Y_train: (16800, 4)
Size of X_test: (3600, 32, 32, 3)
Size of Y_test: (3600, 4)
Size of X_val: (3600, 32, 32, 3)
Size of Y_val: (3600, 4)
```

[**QUESTION**] Why the train/val/test split is important in our case? What is the main interest of the validation data at training time?

[**ANSWER**] CIFAR-4 is a subset of CIFAR-10 with only 4 classes and limited training examples (24,000 pictures). Small datasets are prone to overfitting: The validation set helps detect overfitting and fine-tune the model. The test set ensures unbiased performance evaluation. And splitting the data ensures the model learns effectively while maintaining a reliable measure of generalization and performance.

As for the validation data, it acts as a proxy for unseen data, enabling:

- Model Monitoring: Track the model's generalization ability during training.
- Early Stopping: Stop training if the validation performance stops improving, preventing overfitting.
- Hyperparameter Tuning: Optimize hyperparameters without contaminating the test set.

Without a validation set, you risk overfitting to the training data and cannot effectively assess generalization during training. This split ensures reliable model evaluation and performance improvement.

MULTI-LAYER PERCEPTRON (MLP) MODEL [12 pts]

Analyse of a MLP code [6pts]

It's time to build our first deep neural network...

Below, we provide a small code **THAT IS NOT WORKING**, because something are missing or wrong.

The model is a very simple MLP with only one hidden layer (20 neurons). So we have :

```
x --> "input" layer ==> hidden layer ==> output layer --> ŷ
```

Have a look at the code, and understand how it works.

```
# (1) DEFINE THE ARCHITECTURE OF MY MODEL
#first, I define all the layers and the way they are connected
inputs = tf.keras.Input(shape=(32,32,3)) #my input layer
x = tf.keras.layers.Flatten()(inputs) #cf. question below...
x = tf.keras.layers.Dense(20, activation='relu')(x) #a first hidden layer with 20 neurons
outputs = tf.keras.layers.Dense(4, activation='softmax')(x) # my output layer
#Then, I define my model with the input layer, the output layer and a name
my_mlp_model = tf.keras.Model(inputs=inputs, outputs=outputs, name="my_mlp_model")

#PRINT A SUMMARY OF THE ARCHITECTURE OF MY MODEL WITH THE NUMBER OF TRAINABLE PARAMETERS
my_mlp_model.summary()

# (2) DEFINE THE TRAINING HYPER-PARAMETERS WITH THE "COMPILE" METHOD:
'''
(1) Set the "optimizer" [pick 'adam', 'sgd' or 'rmsprop']
(2) Set the loss [cf. lesson #3, we pick the categorical cross-entropy]
(3) Set the final performance metric to evaluate the model
'''

my_mlp_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# (3) NOW, LET'S TRAIN ON MY DATA WITH THE "FIT" METHOD
'''
(1) Set the number of epochs
(2) Set the size of the (mini)batch
(3) Set the training dataset ==> here, X_train with Y_train
(4) Set the validation dataset (X_val, Y_val)
'''
nb_epochs=4
batch_size=100
training_history = my_mlp_model.fit(X_train,Y_train,
                                    validation_data=(X_val, Y_val),
                                    epochs=nb_epochs,
```

```
                                batch_size=batch_size)

#COMPUTE THE ACCURACY ON THE TRAINING AND TEST SETS
loss_train, acc_train = my_mlp_model.evaluate(X_train, Y_train, batch_size=batch_size)
loss_test, acc_test = my_mlp_model.evaluate(X_test, Y_test, batch_size=batch_size)

print("Performance on the TRAIN set, ACCURACY=",acc_train)
print("Performance on the TEST set, ACCURACY=",acc_test)
```

```
Model: "my_mlp_model"
```

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 32, 32, 3)	0
flatten (Flatten)	(None, 3072)	0
dense (Dense)	(None, 20)	61,460
dense_1 (Dense)	(None, 4)	84

```
 Total params: 61,544 (240.41 KB)
 Trainable params: 61,544 (240.41 KB)
 Non-trainable params: 0 (0.00 B)
Epoch 1/4
168/168 ━━━━━━━━━━━━━━━━━━━━ 3s 6ms/step - accuracy: 0.2533 - loss: 59.8158 - val_accuracy: 0.2406 - val_loss: 1.3865
Epoch 2/4
168/168 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.2536 - loss: 1.3863 - val_accuracy: 0.2447 - val_loss: 1.3865
Epoch 3/4
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.2510 - loss: 1.3864 - val_accuracy: 0.2447 - val_loss: 1.3865
Epoch 4/4
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.2525 - loss: 1.3863 - val_accuracy: 0.2447 - val_loss: 1.3864
168/168 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.2503 - loss: 1.3862
36/36 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.2509 - loss: 1.3863
Performance on the TRAIN set, ACCURACY= 0.2512499988079071
Performance on the TEST set, ACCURACY= 0.2494444400720978
```

[QUESTION]

- (1) What is the activation function for the hidden layer?
- (2) What is the most used activation function in deep learning?
- (3) How many time an image sample will be used during the training?
- (4) How many training iterations (i.e., params update) in total will be processed?

[ANSWER]

(1) The activation function for the hidden layer is relu().

(2) The the most used activation function in deep learning is relu() because ,even with its cons , it is a great way to avoid vanishing gradient and it allows a sparse activation of our network.

(3) Because this deep learning algorithm runs on 4 epochs, each image sample will be used 4 times during the training.

(4) The training dataset have 70% of image samples of our ciphar4 dataset, which means 16800 pictures. Furthermore, the MLP algorithm above is doing a mini-batch Stochastic Gradient Descent with batches of size 100, which means the training dataset have 168 batches. And this algorithm runs on 4 epochs, so in total 672(=4x 24000x0.7 / 100) training iterations will be processed.

[QUESTION]

- (1) Regarding the previous questions, copy and change the code above to make it runs on **10 EPOCHS**. Comment the resulting performance of the model.
- (2) For our MLP, why do we need this line `x = tf.keras.layers.Flatten()(inputs)` ?
- (3) Give the calculation of the number of trainable parameters

```
#Your code here
# (1) DEFINE THE ARCHITECTURE OF MY MODEL
#first, I define all the layers and the way they are connected
inputs = tf.keras.Input(shape=(32,32,3)) #my input layer
x = tf.keras.layers.Flatten()(inputs) #cf. question below...
x = tf.keras.layers.Dense(20, activation='relu')(x) #a first hidden layer with 20 neurons
outputs = tf.keras.layers.Dense(4, activation='softmax')(x) # my output layer
#Then, I define my model with the input layer, the output layer and a name
my_mlp_model = tf.keras.Model(inputs=inputs, outputs=outputs, name="my_mlp_model")

#PRINT A SUMMARY OF THE ARCHITECTURE OF MY MODEL WITH THE NUMBER OF TRAINABLE PARAMETERS
my_mlp_model.summary()

# (2) DEFINE THE TRAINING HYPER-PARAMETERS WITH THE "COMPILE" METHOD:
'''
(1) Set the "optimizer" [pick 'adam', 'sgd' or 'rmsprop']
```

```
(2) Set the loss [cf. lesson #3, we pick the categorical cross-entropy]
(3) Set the final performance metric to evaluate the model
'''

my_mlp_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# (3) NOW, LET'S TRAIN ON MY DATA WITH THE "FIT" METHOD
'''
(1) Set the number of epochs
(2) Set the size of the (mini)batch
(3) Set the training dataset ==> here, X_train with Y_train
(4) Set the validation dataset (X_val, Y_val)
'''
nb_epochs=10
batch_size=100
training_history = my_mlp_model.fit(X_train,Y_train,
                                    validation_data=(X_val, Y_val),
                                    epochs=nb_epochs,
                                    batch_size=batch_size)

#COMPUTE THE ACCURACY ON THE TRAINING AND TEST SETS
loss_train, acc_train = my_mlp_model.evaluate(X_train, Y_train, batch_size=batch_size)
loss_test, acc_test = my_mlp_model.evaluate(X_test, Y_test, batch_size=batch_size)

print("Performance on the TRAIN set, ACCURACY=",acc_train)
print("Performance on the TEST set, ACCURACY=",acc_test)
```

Model: "my_mlp_model"

Layer (type)	Output Shape	Param #
input_layer_6 (InputLayer)	(None, 32, 32, 3)	0
flatten_4 (Flatten)	(None, 3072)	0
dense_12 (Dense)	(None, 20)	61,460
dense_13 (Dense)	(None, 4)	84

Total params: 61,544 (240.41 KB)
Trainable params: 61,544 (240.41 KB)
Non-trainable params: 0 (0.00 B)

```
Epoch 1/10
168/168 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.2566 - loss: 76.4101 - val_accuracy: 0.2406 - val_loss: 1.3866
Epoch 2/10
168/168 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.2492 - loss: 1.3864 - val_accuracy: 0.2406 - val_loss: 1.3865
Epoch 3/10
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.2364 - loss: 1.3864 - val_accuracy: 0.2406 - val_loss: 1.3865
Epoch 4/10
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.2522 - loss: 1.3863 - val_accuracy: 0.2406 - val_loss: 1.3865
Epoch 5/10
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.2487 - loss: 1.3863 - val_accuracy: 0.2447 - val_loss: 1.3864
Epoch 6/10
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.2505 - loss: 1.3863 - val_accuracy: 0.2447 - val_loss: 1.3864
Epoch 7/10
168/168 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.2505 - loss: 1.3863 - val_accuracy: 0.2406 - val_loss: 1.3864
Epoch 8/10
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.2486 - loss: 1.3863 - val_accuracy: 0.2447 - val_loss: 1.3864
Epoch 9/10
168/168 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.2539 - loss: 1.3863 - val_accuracy: 0.2447 - val_loss: 1.3865
Epoch 10/10
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.2563 - loss: 1.3862 - val_accuracy: 0.2406 - val_loss: 1.3864
168/168 ━━━━━━━━━━━━━━━━━━━━ 0s 1ms/step - accuracy: 0.2589 - loss: 1.3862
36/36 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.2522 - loss: 1.3863
Performance on the TRAIN set, ACCURACY= 0.25065475702285767
Performance on the TEST set, ACCURACY= 0.25638890266418457
```

[ANSWER] *Your answer here*

(1) Through all of our 10 epochs, we have an accuracy around 25%. However this classification task has only 4 labels equally distributed in the dataset which means that our model is a good as just choosing randomly a picture. And this accuracy of 25% is the case for both the train and the test set which means that our model is clearly **underfitting**.

(2) The line x = tf.keras.layers.Flatten()(inputs) is essential for preparing the input data for the Dense (fully connected) layers in the MLP architecture.

The input data has the shape (32, 32, 3), representing an RGB image of size 32×32 with 3 color channels. However dense Layers Require 1D Input. This is why we have a Flatten layer (x = tf.keras.layers.Flatten()(inputs)) that converts the 3D input tensor (height, width, channels) into a 1D vector.

(3) We have 2 layers, the first one with 20 neurons and the second one with 4 neurons. Which means that we have 84 (=(20+1)x4) trainable parameters if we count the biases.

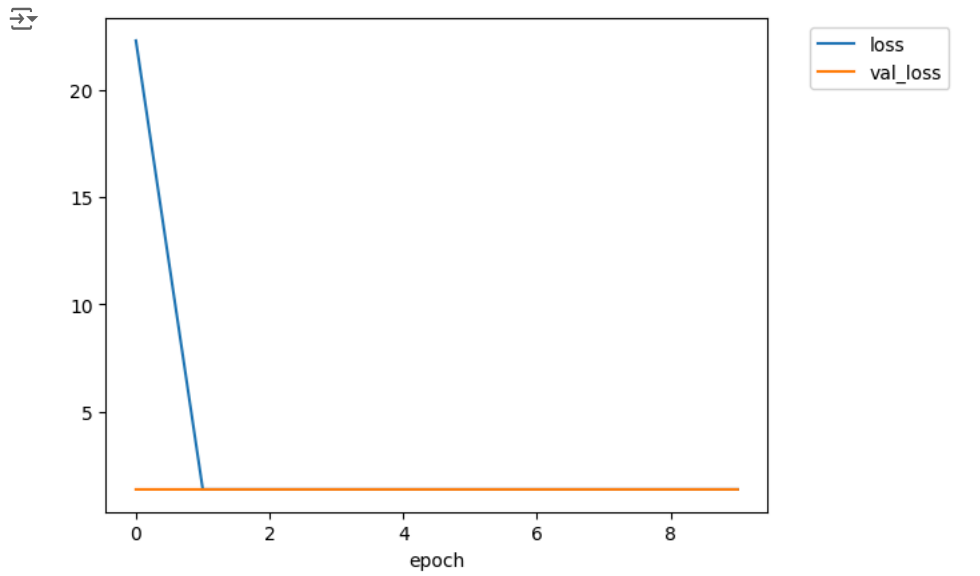
It is always good to have a look on the training curves. That is the role of the "**training_history**" object that we defined in the code as the output of the **fit** method. In this object, we collect all the loss and metric values after each epoch.

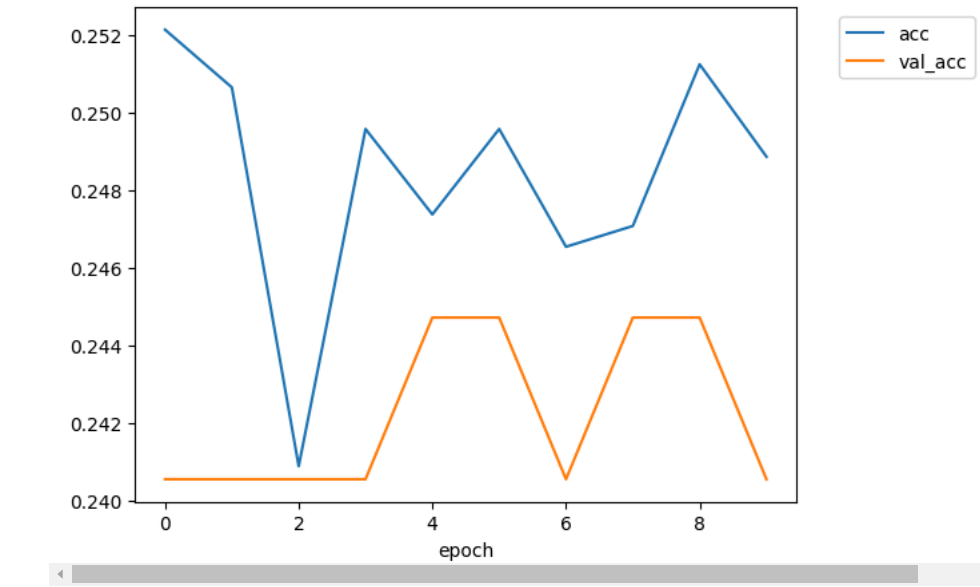
You can use the following **method** to display the train/val curves (loss and accuracy):

```
def display_training_curves(training_history):
  # display loss
  plt.plot(training_history.history['loss'],label='loss')
  plt.plot(training_history.history['val_loss'], label='val_loss')
  plt.xlabel("epoch")
  plt.legend(bbox_to_anchor=(1.05, 1), loc=2)
  plt.show()
  #display accuracy
  plt.plot(training_history.history['accuracy'],label='acc')
  plt.plot(training_history.history['val_accuracy'], label='val_acc')
  plt.xlabel("epoch")
  plt.legend(bbox_to_anchor=(1.05, 1), loc=2)
  plt.show()
```

[QUESTION] Display the training curves from your MLP model.

```
display_training_curves(training_history)
```





Improvements and overfitting [6 pts]

[QUESTION] Use the previous MLP code to build your own model and try to reach a better accuracy performance (e.g., above/around 60%, both train and test)

```
#Your code here

# (1) DEFINE THE ARCHITECTURE OF MY MODEL
#first, I define all the layers and the way they are connected
inputs = tf.keras.Input(shape=(32,32,3)) #my input layer
x = tf.keras.layers.Flatten()(inputs) #cf. question below...
x = tf.keras.layers.Dense(1024, activation='relu')(x) #a first hidden layer with 1024 neurons
x = tf.keras.layers.Dense(768, activation='relu')(x) #a second hidden layer with 768 neurons
x = tf.keras.layers.Dense(512, activation='relu')(x) #a second hidden layer with 512 neurons
x = tf.keras.layers.Dense(256, activation='relu')(x) #a second hidden layer with 256 neurons
x = tf.keras.layers.Dense(128, activation='relu')(x) #a second hidden layer with 128 neurons
outputs = tf.keras.layers.Dense(4, activation='softmax')(x) # my output layer
#Then, I define my model with the input layer, the output layer and a name
my_mlp_model = tf.keras.Model(inputs=inputs, outputs=outputs, name="my_mlp_model")

#PRINT A SUMMARY OF THE ARCHITECTURE OF MY MODEL WITH THE NUMBER OF TRAINABLE PARAMETERS
my_mlp_model.summary()

# (2) DEFINE THE TRAINING HYPER-PARAMETERS WITH THE "COMPILE" METHOD:
'''
(1) Set the "optimizer" [pick 'adam', 'sgd' or 'rmsprop']
(2) Set the loss [cf. lesson #3, we pick the categorical cross-entropy]
(3) Set the final performance metric to evaluate the model
'''

my_mlp_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# (3) NOW, LET'S TRAIN ON MY DATA WITH THE "FIT" METHOD
'''
(1) Set the number of epochs
(2) Set the size of the (mini)batch
(3) Set the training dataset ==> here, X_train with Y_train
(4) Set the validation dataset (X_val, Y_val)
'''
nb_epochs=22
batch_size=100
training_history = my_mlp_model.fit(X_train,Y_train,
                                    validation_data=(X_val, Y_val),
                                    epochs=nb_epochs,
                                    batch_size=batch_size)

#COMPUTE THE ACCURACY ON THE TRAINING AND TEST SETS
loss_train, acc_train = my_mlp_model.evaluate(X_train, Y_train, batch_size=batch_size)
loss_test, acc_test = my_mlp_model.evaluate(X_test, Y_test, batch_size=batch_size)

print("Performance on the TRAIN set, ACCURACY=",acc_train)
print("Performance on the TEST set, ACCURACY=",acc_test)
```

```
Model: "my_mlp_model"
```

Layer (type)	Output Shape	Param #
input_layer_7 (InputLayer)	(None, 32, 32, 3)	0
flatten_5 (Flatten)	(None, 3072)	0
dense_14 (Dense)	(None, 1024)	3,146,752
dense_15 (Dense)	(None, 768)	787,200
dense_16 (Dense)	(None, 512)	393,728
dense_17 (Dense)	(None, 256)	131,328
dense_18 (Dense)	(None, 128)	32,896
dense_19 (Dense)	(None, 4)	516

```
 Total params: 4,492,420 (17.14 MB)
 Trainable params: 4,492,420 (17.14 MB)
 Non-trainable params: 0 (0.00 B)
Epoch 1/22
168/168 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - accuracy: 0.2951 - loss: 98.4515 - val_accuracy: 0.4431 - val_loss: 1.2845
Epoch 2/22
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.4809 - loss: 1.2057 - val_accuracy: 0.5278 - val_loss: 1.1410
Epoch 3/22
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.5181 - loss: 1.1331 - val_accuracy: 0.5417 - val_loss: 1.0966
Epoch 4/22
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.5469 - loss: 1.0766 - val_accuracy: 0.5264 - val_loss: 1.1308
Epoch 5/22
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.5730 - loss: 1.0358 - val_accuracy: 0.5669 - val_loss: 1.0396
Epoch 6/22
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.5634 - loss: 1.0461 - val_accuracy: 0.5669 - val_loss: 1.0457
Epoch 7/22
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.5721 - loss: 1.0150 - val_accuracy: 0.5322 - val_loss: 1.1097
Epoch 8/22
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.5710 - loss: 1.0289 - val_accuracy: 0.5853 - val_loss: 1.0232
Epoch 9/22
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.6026 - loss: 0.9538 - val_accuracy: 0.5783 - val_loss: 1.0148
Epoch 10/22
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.6018 - loss: 0.9657 - val_accuracy: 0.5736 - val_loss: 1.0351
Epoch 11/22
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.5959 - loss: 0.9706 - val_accuracy: 0.5908 - val_loss: 0.9961
Epoch 12/22
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6139 - loss: 0.9473 - val_accuracy: 0.5869 - val_loss: 1.0101
Epoch 13/22
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6192 - loss: 0.9190 - val_accuracy: 0.5556 - val_loss: 1.0578
Epoch 14/22
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.5969 - loss: 0.9774 - val_accuracy: 0.5881 - val_loss: 1.0106
Epoch 15/22
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6263 - loss: 0.9128 - val_accuracy: 0.5628 - val_loss: 1.0635
Epoch 16/22
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.6236 - loss: 0.9174 - val_accuracy: 0.6083 - val_loss: 0.9879
Epoch 17/22
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6240 - loss: 0.9064 - val_accuracy: 0.6003 - val_loss: 1.0023
Epoch 18/22
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6222 - loss: 0.9166 - val_accuracy: 0.5853 - val_loss: 1.0323
Epoch 19/22
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6211 - loss: 0.9128 - val_accuracy: 0.5472 - val_loss: 1.0855
Epoch 20/22
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.6299 - loss: 0.9156 - val_accuracy: 0.5975 - val_loss: 1.0084
Epoch 21/22
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6308 - loss: 0.9071 - val_accuracy: 0.6200 - val_loss: 0.9768
Epoch 22/22
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6438 - loss: 0.8654 - val_accuracy: 0.5719 - val_loss: 1.0340
168/168 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.6338 - loss: 0.8785
36/36 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5597 - loss: 1.0570
Performance on the TRAIN set, ACCURACY= 0.6311904788017273
Performance on the TEST set, ACCURACY= 0.5616666674613953
```

[QUESTION] For illustration/educational purpose, use the code of your last model and adapt it so that your model clearly **OVERFITS**.

We need to see the overfitting issue on the training curve!

(Think about the reasons of overfitting?)

```
#Your code here

# (1) DEFINE THE ARCHITECTURE OF MY MODEL
#first, I define all the layers and the way they are connected
inputs = tf.keras.Input(shape=(32,32,3)) #my input layer
x = tf.keras.layers.Flatten()(inputs) #cf. question below...
x = tf.keras.layers.Dense(1024, activation='relu')(x) #a first hidden layer with 1024 neurons
x = tf.keras.layers.Dense(768, activation='relu')(x) #a second hidden layer with 768 neurons
x = tf.keras.layers.Dense(512, activation='relu')(x) #a second hidden layer with 512 neurons
```

```
x = tf.keras.layers.Dense(256, activation='relu')(x) #a second hidden layer with 256 neurons
x = tf.keras.layers.Dense(128, activation='relu')(x) #a second hidden layer with 128 neurons
outputs = tf.keras.layers.Dense(4, activation='softmax')(x) # my output layer
#Then, I define my model with the input layer, the output layer and a name
my_mlp_model = tf.keras.Model(inputs=inputs, outputs=outputs, name="my_mlp_model")

#PRINT A SUMMARY OF THE ARCHITECTURE OF MY MODEL WITH THE NUMBER OF TRAINABLE PARAMETERS
my_mlp_model.summary()

# (2) DEFINE THE TRAINING HYPER-PARAMETERS WITH THE "COMPILE" METHOD:
'''
(1) Set the "optimizer" [pick 'adam', 'sgd' or 'rmsprop']
(2) Set the loss [cf. lesson #3, we pick the categorical cross-entropy]
(3) Set the final performance metric to evaluate the model
'''

my_mlp_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# (3) NOW, LET'S TRAIN ON MY DATA WITH THE "FIT" METHOD
'''
(1) Set the number of epochs
(2) Set the size of the (mini)batch
(3) Set the training dataset ==> here, X_train with Y_train
(4) Set the validation dataset (X_val, Y_val)
'''
nb_epochs=60
batch_size=100
training_history = my_mlp_model.fit(X_train,Y_train,
                                    validation_data=(X_val, Y_val),
                                    epochs=nb_epochs,
                                    batch_size=batch_size)

#COMPUTE THE ACCURACY ON THE TRAINING AND TEST SETS
loss_train, acc_train = my_mlp_model.evaluate(X_train, Y_train, batch_size=batch_size)
loss_test, acc_test = my_mlp_model.evaluate(X_test, Y_test, batch_size=batch_size)

print("Performance on the TRAIN set, ACCURACY=",acc_train)
print("Performance on the TEST set, ACCURACY=",acc_test)

display_training_curves(training_history)
```

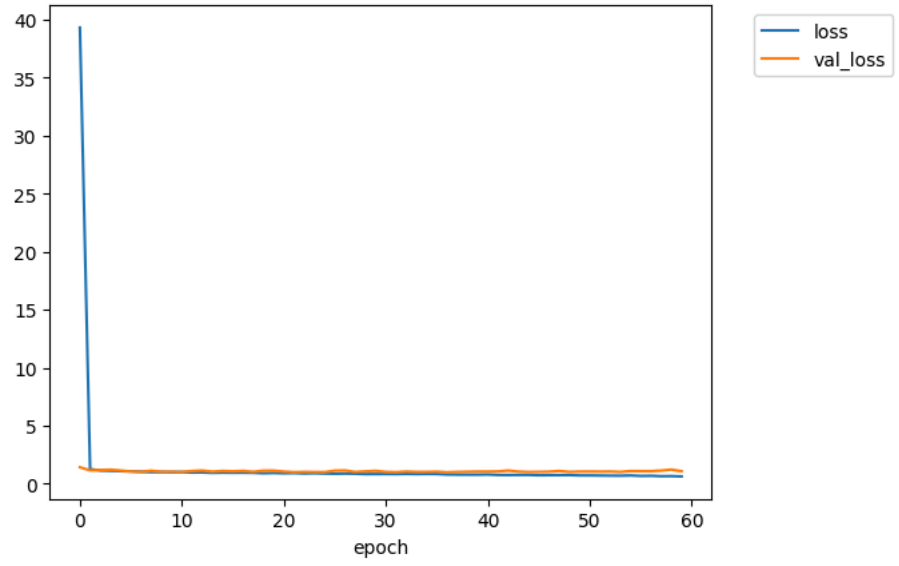
Model: "my_mlp_model"

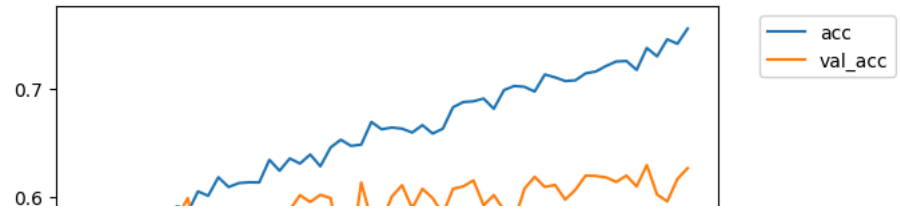
Layer (type)	Output Shape	Param #
input_layer_8 (InputLayer)	(None, 32, 32, 3)	0
flatten_6 (Flatten)	(None, 3072)	0
dense_20 (Dense)	(None, 1024)	3,146,752
dense_21 (Dense)	(None, 768)	787,200
dense_22 (Dense)	(None, 512)	393,728
dense_23 (Dense)	(None, 256)	131,328
dense_24 (Dense)	(None, 128)	32,896
dense_25 (Dense)	(None, 4)	516

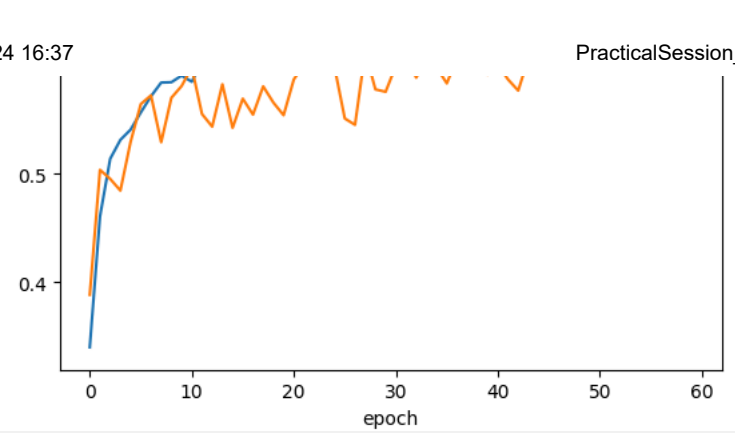
Total params: 4,492,420 (17.14 MB)
Trainable params: 4,492,420 (17.14 MB)
Non-trainable params: 0 (0.00 B)

```
Epoch 1/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 4s 8ms/step - accuracy: 0.2847 - loss: 119.3022 - val_accuracy: 0.3883 - val_loss: 1.4112
Epoch 2/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.4485 - loss: 1.2745 - val_accuracy: 0.5039 - val_loss: 1.1479
Epoch 3/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.5182 - loss: 1.1309 - val_accuracy: 0.4956 - val_loss: 1.1697
Epoch 4/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.5228 - loss: 1.1212 - val_accuracy: 0.4847 - val_loss: 1.2066
Epoch 5/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.5377 - loss: 1.0870 - val_accuracy: 0.5297 - val_loss: 1.1188
Epoch 6/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.5569 - loss: 1.0568 - val_accuracy: 0.5647 - val_loss: 1.0615
Epoch 7/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.5742 - loss: 1.0196 - val_accuracy: 0.5728 - val_loss: 1.0252
Epoch 8/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.5850 - loss: 0.9861 - val_accuracy: 0.5294 - val_loss: 1.1151
Epoch 9/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.5866 - loss: 0.9986 - val_accuracy: 0.5706 - val_loss: 1.0250
Epoch 10/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.5935 - loss: 0.9896 - val_accuracy: 0.5814 - val_loss: 1.0342
Epoch 11/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.5902 - loss: 0.9808 - val_accuracy: 0.5986 - val_loss: 1.0103
Epoch 12/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6075 - loss: 0.9495 - val_accuracy: 0.5556 - val_loss: 1.0882
Epoch 13/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6053 - loss: 0.9440 - val_accuracy: 0.5439 - val_loss: 1.1298
Epoch 14/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6108 - loss: 0.9511 - val_accuracy: 0.5831 - val_loss: 1.0380
Epoch 15/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6116 - loss: 0.9466 - val_accuracy: 0.5428 - val_loss: 1.0931
Epoch 16/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6128 - loss: 0.9419 - val_accuracy: 0.5697 - val_loss: 1.0643
Epoch 17/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6226 - loss: 0.9295 - val_accuracy: 0.5550 - val_loss: 1.1019
Epoch 18/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6077 - loss: 0.9410 - val_accuracy: 0.5811 - val_loss: 1.0243
Epoch 19/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6377 - loss: 0.8977 - val_accuracy: 0.5661 - val_loss: 1.1208
Epoch 20/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.6346 - loss: 0.8932 - val_accuracy: 0.5544 - val_loss: 1.1217
Epoch 21/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.6388 - loss: 0.8934 - val_accuracy: 0.5875 - val_loss: 1.0414
Epoch 22/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.6408 - loss: 0.8846 - val_accuracy: 0.6014 - val_loss: 0.9890
Epoch 23/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.6357 - loss: 0.8809 - val_accuracy: 0.5950 - val_loss: 1.0110
Epoch 24/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.6357 - loss: 0.8792 - val_accuracy: 0.6017 - val_loss: 0.9995
Epoch 25/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6500 - loss: 0.8552 - val_accuracy: 0.5986 - val_loss: 0.9936
Epoch 26/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6582 - loss: 0.8403 - val_accuracy: 0.5514 - val_loss: 1.1204
Epoch 27/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.6613 - loss: 0.8385 - val_accuracy: 0.5456 - val_loss: 1.1353
Epoch 28/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6317 - loss: 0.8884 - val_accuracy: 0.6131 - val_loss: 1.0070
Epoch 29/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6627 - loss: 0.8338 - val_accuracy: 0.5783 - val_loss: 1.0610
Epoch 30/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6595 - loss: 0.8340 - val_accuracy: 0.5761 - val_loss: 1.1023
Epoch 31/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6667 - loss: 0.8288 - val_accuracy: 0.6000 - val_loss: 1.0120
Epoch 32/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6672 - loss: 0.8116 - val_accuracy: 0.6106 - val_loss: 0.9890
Epoch 33/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6731 - loss: 0.8134 - val_accuracy: 0.5892 - val_loss: 1.0427
Epoch 34/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6680 - loss: 0.8231 - val_accuracy: 0.6072 - val_loss: 1.0132
```

```
Epoch 35/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6673 - loss: 0.8223 - val_accuracy: 0.5986 - val_loss: 1.0118
Epoch 36/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6678 - loss: 0.8194 - val_accuracy: 0.5839 - val_loss: 1.0335
Epoch 37/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6913 - loss: 0.7650 - val_accuracy: 0.6072 - val_loss: 0.9784
Epoch 38/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6995 - loss: 0.7473 - val_accuracy: 0.6094 - val_loss: 1.0100
Epoch 39/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.6953 - loss: 0.7567 - val_accuracy: 0.6150 - val_loss: 1.0258
Epoch 40/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.7031 - loss: 0.7548 - val_accuracy: 0.5919 - val_loss: 1.0451
Epoch 41/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.6857 - loss: 0.7826 - val_accuracy: 0.6014 - val_loss: 1.0454
Epoch 42/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.7050 - loss: 0.7300 - val_accuracy: 0.5875 - val_loss: 1.0545
Epoch 43/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.7100 - loss: 0.7265 - val_accuracy: 0.5772 - val_loss: 1.1260
Epoch 44/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.7026 - loss: 0.7420 - val_accuracy: 0.6072 - val_loss: 1.0386
Epoch 45/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6989 - loss: 0.7441 - val_accuracy: 0.6183 - val_loss: 1.0064
Epoch 46/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.7189 - loss: 0.7108 - val_accuracy: 0.6089 - val_loss: 1.0161
Epoch 47/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.7244 - loss: 0.6989 - val_accuracy: 0.6108 - val_loss: 1.0338
Epoch 48/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.7158 - loss: 0.7097 - val_accuracy: 0.5972 - val_loss: 1.0899
Epoch 49/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.7179 - loss: 0.7129 - val_accuracy: 0.6064 - val_loss: 1.0159
Epoch 50/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.7252 - loss: 0.6833 - val_accuracy: 0.6194 - val_loss: 1.0414
Epoch 51/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.7149 - loss: 0.7082 - val_accuracy: 0.6192 - val_loss: 1.0478
Epoch 52/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.7293 - loss: 0.6809 - val_accuracy: 0.6178 - val_loss: 1.0380
Epoch 53/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.7309 - loss: 0.6757 - val_accuracy: 0.6136 - val_loss: 1.0461
Epoch 54/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.7376 - loss: 0.6584 - val_accuracy: 0.6194 - val_loss: 1.0126
Epoch 55/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.7336 - loss: 0.6770 - val_accuracy: 0.6094 - val_loss: 1.0848
Epoch 56/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.7454 - loss: 0.6537 - val_accuracy: 0.6292 - val_loss: 1.0800
Epoch 57/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.7428 - loss: 0.6455 - val_accuracy: 0.6019 - val_loss: 1.0769
Epoch 58/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.7446 - loss: 0.6442 - val_accuracy: 0.5956 - val_loss: 1.1276
Epoch 59/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.7339 - loss: 0.6679 - val_accuracy: 0.6164 - val_loss: 1.2128
Epoch 60/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.7638 - loss: 0.6027 - val_accuracy: 0.6264 - val_loss: 1.0702
168/168 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.7637 - loss: 0.6055
36/36 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.6150 - loss: 1.0940
Performance on the TRAIN set, ACCURACY= 0.7672619223594666
Performance on the TEST set, ACCURACY= 0.6113888621330261
```






0.5
0.4
0
10
20
30
40
50
60
epoch

[QUESTION] Try to find an optimal architecture without overfitting by using a regularization (or other) technique of your choice. The goal is to have a performance > 60% without overfitting after **20** epochs.

```
#Your code here

# (1) DEFINE THE ARCHITECTURE OF MY MODEL
#first, I define all the layers and the way they are connected
inputs = tf.keras.Input(shape=(32,32,3)) #my input layer
x = tf.keras.layers.Flatten()(inputs) #cf. question below...

#x = tf.keras.layers.Dense(1024, activation='relu',kernel_regularizer=tf.keras.regularizers.l2(0.01))(x) does not work when doing it on
#x = tf.keras.layers.BatchNormalization()(x) : ultra overfitting (training_acc : 95% and test_accuracy : 60%)
#x = tf.keras.layers.Dropout(0.1)(x)  Dropping 10% of neurons on the first layer only, allows to eliminates an important part of our pr

x = tf.keras.layers.Dense(1024, activation='relu')(x) #a first hidden layer with 1024 neurons
x = tf.keras.layers.Dropout(0.1)(x)  #Drop 10% of neurons on the first layer
x = tf.keras.layers.Dense(768, activation='relu')(x) #a second hidden layer with 768 neurons
x = tf.keras.layers.Dense(512, activation='relu')(x) #a second hidden layer with 512 neurons
x = tf.keras.layers.Dense(256, activation='relu')(x) #a second hidden layer with 256 neurons
x = tf.keras.layers.Dense(128, activation='relu')(x) #a second hidden layer with 128 neurons
outputs = tf.keras.layers.Dense(4, activation='softmax')(x) # my output layer
#Then, I define my model with the input layer, the output layer and a name
my_mlp_model = tf.keras.Model(inputs=inputs, outputs=outputs, name="my_mlp_model")

#PRINT A SUMMARY OF THE ARCHITECTURE OF MY MODEL WITH THE NUMBER OF TRAINABLE PARAMETERS
my_mlp_model.summary()

# (2) DEFINE THE TRAINING HYPER-PARAMETERS WITH THE "COMPILE" METHOD:
'''
(1) Set the "optimizer" [pick 'adam', 'sgd' or 'rmsprop']
(2) Set the loss [cf. lesson #3, we pick the categorical cross-entropy]
(3) Set the final performance metric to evaluate the model
'''

my_mlp_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# (3) NOW, LET'S TRAIN ON MY DATA WITH THE "FIT" METHOD
'''
(1) Set the number of epochs
(2) Set the size of the (mini)batch
(3) Set the training dataset ==> here, X_train with Y_train
(4) Set the validation dataset (X_val, Y_val)
'''
nb_epochs=60
batch_size=100
training_history = my_mlp_model.fit(X_train,Y_train,
                                    validation_data=(X_val, Y_val),
                                    epochs=nb_epochs,
                                    batch_size=batch_size)

#COMPUTE THE ACCURACY ON THE TRAINING AND TEST SETS
loss_train, acc_train = my_mlp_model.evaluate(X_train, Y_train, batch_size=batch_size)
loss_test, acc_test = my_mlp_model.evaluate(X_test, Y_test, batch_size=batch_size)

print("Performance on the TRAIN set, ACCURACY=",acc_train)
print("Performance on the TEST set, ACCURACY=",acc_test)
display_training_curves(training_history)
```

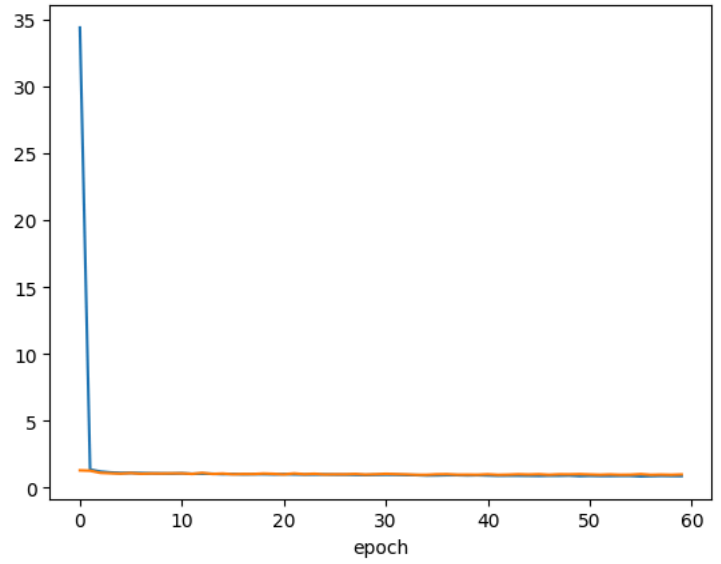
Model: "my_mlp_model"

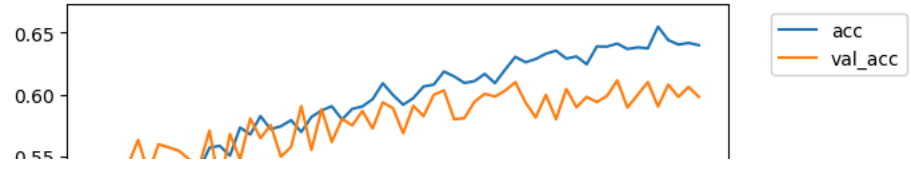
Layer (type)	Output Shape	Param #
input_layer_11 (InputLayer)	(None, 32, 32, 3)	0
flatten_9 (Flatten)	(None, 3072)	0
dense_38 (Dense)	(None, 1024)	3,146,752
dropout_2 (Dropout)	(None, 1024)	0
dense_39 (Dense)	(None, 768)	787,200
dense_40 (Dense)	(None, 512)	393,728
dense_41 (Dense)	(None, 256)	131,328
dense_42 (Dense)	(None, 128)	32,896
dense_43 (Dense)	(None, 4)	516

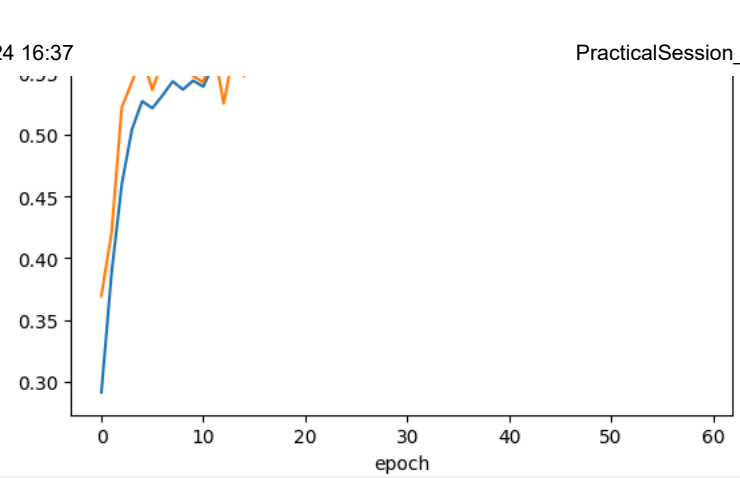
Total params: 4,492,420 (17.14 MB)
Trainable params: 4,492,420 (17.14 MB)
Non-trainable params: 0 (0.00 B)

```
Epoch 1/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 5s 11ms/step - accuracy: 0.2687 - loss: 119.5292 - val_accuracy: 0.3694 - val_loss: 1.3006
Epoch 2/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 2s 4ms/step - accuracy: 0.3617 - loss: 1.4203 - val_accuracy: 0.4206 - val_loss: 1.2740
Epoch 3/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.4427 - loss: 1.2367 - val_accuracy: 0.5222 - val_loss: 1.1156
Epoch 4/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.5040 - loss: 1.1457 - val_accuracy: 0.5428 - val_loss: 1.0827
Epoch 5/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.5206 - loss: 1.1149 - val_accuracy: 0.5633 - val_loss: 1.0586
Epoch 6/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.5273 - loss: 1.1056 - val_accuracy: 0.5364 - val_loss: 1.0980
Epoch 7/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.5273 - loss: 1.1056 - val_accuracy: 0.5597 - val_loss: 1.0470
Epoch 8/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.5454 - loss: 1.0775 - val_accuracy: 0.5572 - val_loss: 1.0689
Epoch 9/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.5352 - loss: 1.0753 - val_accuracy: 0.5544 - val_loss: 1.0647
Epoch 10/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.5474 - loss: 1.0588 - val_accuracy: 0.5475 - val_loss: 1.0769
Epoch 11/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.5414 - loss: 1.0716 - val_accuracy: 0.5425 - val_loss: 1.0772
Epoch 12/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.5524 - loss: 1.0403 - val_accuracy: 0.5708 - val_loss: 1.0368
Epoch 13/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.5559 - loss: 1.0546 - val_accuracy: 0.5253 - val_loss: 1.1143
Epoch 14/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.5446 - loss: 1.0552 - val_accuracy: 0.5681 - val_loss: 1.0334
Epoch 15/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.5806 - loss: 1.0059 - val_accuracy: 0.5475 - val_loss: 1.0619
Epoch 16/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.5658 - loss: 1.0288 - val_accuracy: 0.5806 - val_loss: 1.0083
Epoch 17/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.5819 - loss: 1.0005 - val_accuracy: 0.5647 - val_loss: 1.0294
Epoch 18/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.5742 - loss: 1.0033 - val_accuracy: 0.5753 - val_loss: 1.0250
Epoch 19/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.5853 - loss: 0.9875 - val_accuracy: 0.5497 - val_loss: 1.0627
Epoch 20/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.5844 - loss: 0.9882 - val_accuracy: 0.5575 - val_loss: 1.0409
Epoch 21/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.5598 - loss: 1.0279 - val_accuracy: 0.5906 - val_loss: 0.9992
Epoch 22/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.5930 - loss: 0.9764 - val_accuracy: 0.5553 - val_loss: 1.0715
Epoch 23/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.5830 - loss: 0.9980 - val_accuracy: 0.5881 - val_loss: 1.0117
Epoch 24/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.5928 - loss: 0.9759 - val_accuracy: 0.5617 - val_loss: 1.0450
Epoch 25/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.5814 - loss: 0.9914 - val_accuracy: 0.5803 - val_loss: 0.9958
Epoch 26/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.5968 - loss: 0.9685 - val_accuracy: 0.5750 - val_loss: 1.0047
Epoch 27/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.5936 - loss: 0.9658 - val_accuracy: 0.5867 - val_loss: 1.0057
Epoch 28/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.5936 - loss: 0.9670 - val_accuracy: 0.5725 - val_loss: 1.0314
Epoch 29/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6178 - loss: 0.9396 - val_accuracy: 0.5936 - val_loss: 0.9883
Epoch 30/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6094 - loss: 0.9454 - val_accuracy: 0.5889 - val_loss: 1.0120
Epoch 31/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.5956 - loss: 0.9559 - val_accuracy: 0.5686 - val_loss: 1.0411
Epoch 32/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.5960 - loss: 0.9706 - val_accuracy: 0.5908 - val_loss: 1.0185
Epoch 33/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6113 - loss: 0.9395 - val_accuracy: 0.5825 - val_loss: 1.0005
```

```
Epoch 34/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6115 - loss: 0.9388 - val_accuracy: 0.5997 - val_loss: 0.9695
Epoch 35/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6116 - loss: 0.9320 - val_accuracy: 0.6033 - val_loss: 0.9685
Epoch 36/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.6237 - loss: 0.9137 - val_accuracy: 0.5800 - val_loss: 1.0091
Epoch 37/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.6089 - loss: 0.9449 - val_accuracy: 0.5808 - val_loss: 1.0166
Epoch 38/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.6084 - loss: 0.9446 - val_accuracy: 0.5942 - val_loss: 0.9789
Epoch 39/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6248 - loss: 0.9112 - val_accuracy: 0.6006 - val_loss: 0.9828
Epoch 40/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6110 - loss: 0.9341 - val_accuracy: 0.5983 - val_loss: 0.9754
Epoch 41/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6233 - loss: 0.9077 - val_accuracy: 0.6031 - val_loss: 1.0095
Epoch 42/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6334 - loss: 0.8893 - val_accuracy: 0.6100 - val_loss: 0.9674
Epoch 43/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6338 - loss: 0.8798 - val_accuracy: 0.5936 - val_loss: 0.9816
Epoch 44/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6288 - loss: 0.8955 - val_accuracy: 0.5814 - val_loss: 1.0120
Epoch 45/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6368 - loss: 0.8893 - val_accuracy: 0.5997 - val_loss: 0.9915
Epoch 46/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6355 - loss: 0.8918 - val_accuracy: 0.5800 - val_loss: 1.0147
Epoch 47/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6281 - loss: 0.8967 - val_accuracy: 0.6044 - val_loss: 0.9728
Epoch 48/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6340 - loss: 0.8809 - val_accuracy: 0.5897 - val_loss: 1.0087
Epoch 49/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6145 - loss: 0.9282 - val_accuracy: 0.5981 - val_loss: 1.0048
Epoch 50/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.6370 - loss: 0.8674 - val_accuracy: 0.5939 - val_loss: 1.0208
Epoch 51/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6396 - loss: 0.8808 - val_accuracy: 0.5989 - val_loss: 0.9951
Epoch 52/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6436 - loss: 0.8707 - val_accuracy: 0.6114 - val_loss: 0.9733
Epoch 53/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6424 - loss: 0.8657 - val_accuracy: 0.5894 - val_loss: 0.9974
Epoch 54/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.6419 - loss: 0.8788 - val_accuracy: 0.5997 - val_loss: 0.9708
Epoch 55/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.6360 - loss: 0.8898 - val_accuracy: 0.6100 - val_loss: 0.9790
Epoch 56/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.6500 - loss: 0.8553 - val_accuracy: 0.5903 - val_loss: 1.0230
Epoch 57/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.6467 - loss: 0.8647 - val_accuracy: 0.6081 - val_loss: 0.9667
Epoch 58/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6463 - loss: 0.8741 - val_accuracy: 0.5981 - val_loss: 0.9855
Epoch 59/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6439 - loss: 0.8718 - val_accuracy: 0.6061 - val_loss: 0.9689
Epoch 60/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6432 - loss: 0.8592 - val_accuracy: 0.5981 - val_loss: 0.9956
168/168 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.6670 - loss: 0.8248
36/36 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.5836 - loss: 1.0136
Performance on the TRAIN set, ACCURACY= 0.6651190519332886
Performance on the TEST set, ACCURACY= 0.5869444608688354
```






0.50
0.45
0.40
0.35
0.30
0
10
20
30
40
50
60
epoch

CONVOLUTIONAL NEURAL NETWORK (CNN) [12 pts]

Build a first architecture [6 pts]

MLPs are great but CNNs should work better for our image classification problem...

For that, we will use new layers from TF.KERAS:

- tf.keras.layers.Conv2D() an example is:

```
l = tf.keras.layers.Conv2D(32,kernel_size=3,activation='relu')(l_input)
```

  Here, we ask for 32 convolutional kernels of size [3,3]. By default the stride is set to '1' and the padding is 'valid'.

- tf.keras.layers.MaxPooling2D() an example is:

```
l = tf.keras.layers.MaxPooling2D()(l_input)
```

  By default the stride is set to '2' and the padding is 'valid'.

[QUESTION] With the default parameters of *Conv2D()*, do you expect to have the same shape for the output tensor?

[ANSWER] Not at all, because of the padding='valid' (so no padding added around the input) by default, the output tensor will at least shrink by 1 in height and width.

[QUESTION] With the default parameters of *MaxPooling2D()*, what do you expect on the shape of the output tensor?

[ANSWER] With those default parameters, after a MaxPooling2D(), the output tensor will have the integer part of the half of both the height and width of the input tensor.

[QUESTION] Try to build a first CNN model with this architecture:

```
x --> Conv2D (32 filters) ==> MaxPooling ==> Conv2D (64 filters) ==> MaxPooling ==> Flatten ==> Dense (4) --> ŷ
```

```
#Your code here

# (1) DEFINE THE ARCHITECTURE OF MY MODEL
#first, I define all the layers and the way they are connected
inputs = tf.keras.Input(shape=(32,32,3)) #my input layer

x = tf.keras.layers.Conv2D(32,kernel_size=3,activation='relu')(inputs)
x = tf.keras.layers.MaxPooling2D()(x)
x = tf.keras.layers.Conv2D(64,kernel_size=3,activation='relu')(x)
x = tf.keras.layers.MaxPooling2D()(x)

x = tf.keras.layers.Flatten()(x) #cf. question below...

outputs = tf.keras.layers.Dense(4, activation='softmax')(x) # my output layer
#Then, I define my model with the input layer, the output layer and a name
my_cnn_model = tf.keras.Model(inputs=inputs, outputs=outputs, name="my_cnn_model")

#PRINT A SUMMARY OF THE ARCHITECTURE OF MY MODEL WITH THE NUMBER OF TRAINABLE PARAMETERS
my_cnn_model.summary()

# (2) DEFINE THE TRAINING HYPER-PARAMETERS WITH THE "COMPILE" METHOD:
'''
(1) Set the "optimizer" [pick 'adam', 'sgd' or 'rmsprop']
(2) Set the loss [cf. lesson #3, we pick the categorical cross-entropy]
(3) Set the final performance metric to evaluate the model
'''

my_cnn_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# (3) NOW, LET'S TRAIN ON MY DATA WITH THE "FIT" METHOD
'''
(1) Set the number of epochs
(2) Set the size of the (mini)batch
(3) Set the training dataset ==> here, X_train with Y_train
```

```
(4) Set the validation dataset (X_val, Y_val)
'''
nb_epochs=7
batch_size=100
training_history = my_cnn_model.fit(X_train,Y_train,
                                    validation_data=(X_val, Y_val),
                                    epochs=nb_epochs,
                                    batch_size=batch_size)

#COMPUTE THE ACCURACY ON THE TRAINING AND TEST SETS
loss_train, acc_train = my_cnn_model.evaluate(X_train, Y_train, batch_size=batch_size)
loss_test, acc_test = my_cnn_model.evaluate(X_test, Y_test, batch_size=batch_size)

print("Performance on the TRAIN set, ACCURACY=",acc_train)
print("Performance on the TEST set, ACCURACY=",acc_test)
```

Model: "my_cnn_model"

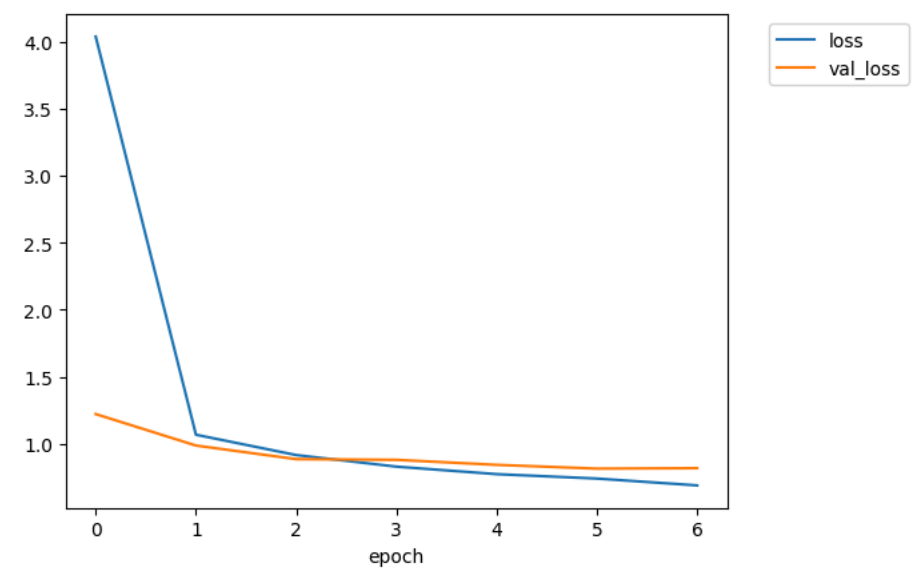
Layer (type)	Output Shape	Param #
input_layer_12 (InputLayer)	(None, 32, 32, 3)	0
conv2d_4 (Conv2D)	(None, 30, 30, 32)	896
max_pooling2d_4 (MaxPooling2D)	(None, 15, 15, 32)	0
conv2d_5 (Conv2D)	(None, 13, 13, 64)	18,496
max_pooling2d_5 (MaxPooling2D)	(None, 6, 6, 64)	0
flatten_10 (Flatten)	(None, 2304)	0
dense_44 (Dense)	(None, 4)	9,220

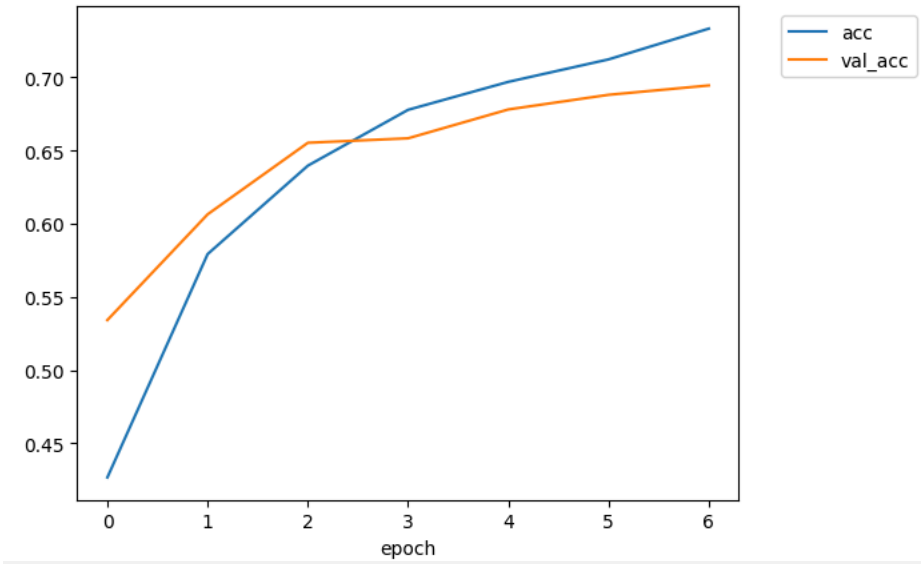
Total params: 28,612 (111.77 KB)
Trainable params: 28,612 (111.77 KB)
Non-trainable params: 0 (0.00 B)

```
Epoch 1/7
168/168 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.3539 - loss: 10.3783 - val_accuracy: 0.5342 - val_loss: 1.2216
Epoch 2/7
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.5634 - loss: 1.1031 - val_accuracy: 0.6064 - val_loss: 0.9867
Epoch 3/7
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.6311 - loss: 0.9360 - val_accuracy: 0.6553 - val_loss: 0.8853
Epoch 4/7
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6785 - loss: 0.8200 - val_accuracy: 0.6583 - val_loss: 0.8801
Epoch 5/7
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.7027 - loss: 0.7615 - val_accuracy: 0.6781 - val_loss: 0.8426
Epoch 6/7
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.7157 - loss: 0.7308 - val_accuracy: 0.6881 - val_loss: 0.8145
Epoch 7/7
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.7418 - loss: 0.6742 - val_accuracy: 0.6944 - val_loss: 0.8179
168/168 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7625 - loss: 0.6317
36/36 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7043 - loss: 0.8290
Performance on the TRAIN set, ACCURACY= 0.7643452286720276
Performance on the TEST set, ACCURACY= 0.694444179534912
```

[QUESTION] Display the training curves from your CNN model.

```
#Your code here
display_training_curves(training_history)
```





Improvements and overfitting [6 pts]

[QUESTION] Use, the code of your last model and adapt it so that your model **OVERFITS**.

We need to see the overfitting issue on the training curve!

```
#Your code here

# (1) DEFINE THE ARCHITECTURE OF MY MODEL
#first, I define all the layers and the way they are connected
inputs = tf.keras.Input(shape=(32,32,3)) #my input layer

x = tf.keras.layers.Conv2D(32,kernel_size=3,activation='relu')(inputs)
x = tf.keras.layers.MaxPooling2D()(x)
x = tf.keras.layers.Conv2D(64,kernel_size=3,activation='relu')(x)
x = tf.keras.layers.MaxPooling2D()(x)

x = tf.keras.layers.Flatten()(x) #cf. question below...

outputs = tf.keras.layers.Dense(4, activation='softmax')(x) # my output layer
#Then, I define my model with the input layer, the output layer and a name
my_cnn_model = tf.keras.Model(inputs=inputs, outputs=outputs, name="my_cnn_model")

#PRINT A SUMMARY OF THE ARCHITECTURE OF MY MODEL WITH THE NUMBER OF TRAINABLE PARAMETERS
my_cnn_model.summary()

# (2) DEFINE THE TRAINING HYPER-PARAMETERS WITH THE "COMPILE" METHOD:
'''
(1) Set the "optimizer" [pick 'adam', 'sgd' or 'rmsprop']
(2) Set the loss [cf. lesson #3, we pick the categorical cross-entropy]
(3) Set the final performance metric to evaluate the model
'''
```

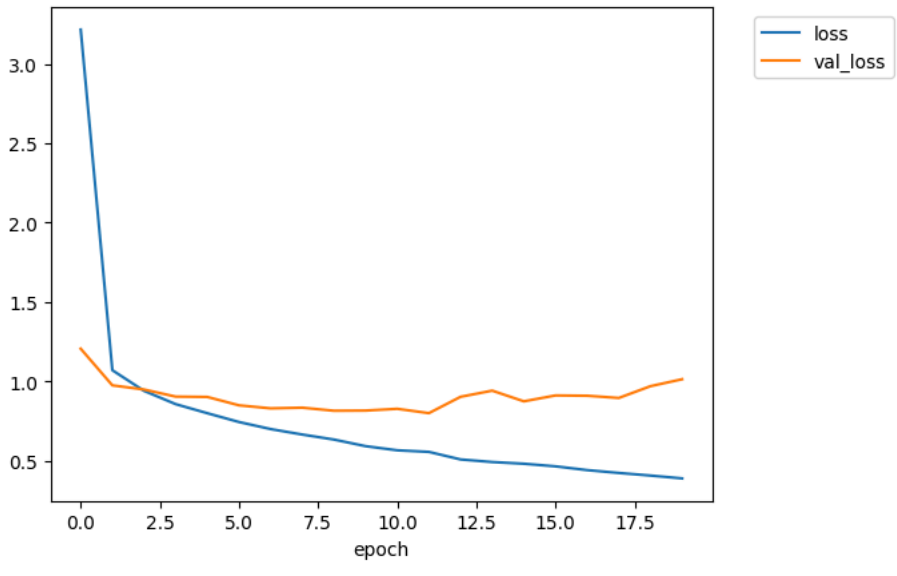
```
my_cnn_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# (3) NOW, LET'S TRAIN ON MY DATA WITH THE "FIT" METHOD
'''
(1) Set the number of epochs
(2) Set the size of the (mini)batch
(3) Set the training dataset ==> here, X_train with Y_train
(4) Set the validation dataset (X_val, Y_val)
'''
nb_epochs=20
batch_size=100
training_history = my_cnn_model.fit(X_train,Y_train,
                                    validation_data=(X_val, Y_val),
                                    epochs=nb_epochs,
                                    batch_size=batch_size)

#COMPUTE THE ACCURACY ON THE TRAINING AND TEST SETS
loss_train, acc_train = my_cnn_model.evaluate(X_train, Y_train, batch_size=batch_size)
loss_test, acc_test = my_cnn_model.evaluate(X_test, Y_test, batch_size=batch_size)

print("Performance on the TRAIN set, ACCURACY=",acc_train)
print("Performance on the TEST set, ACCURACY=",acc_test)

display_training_curves(training_history)
```

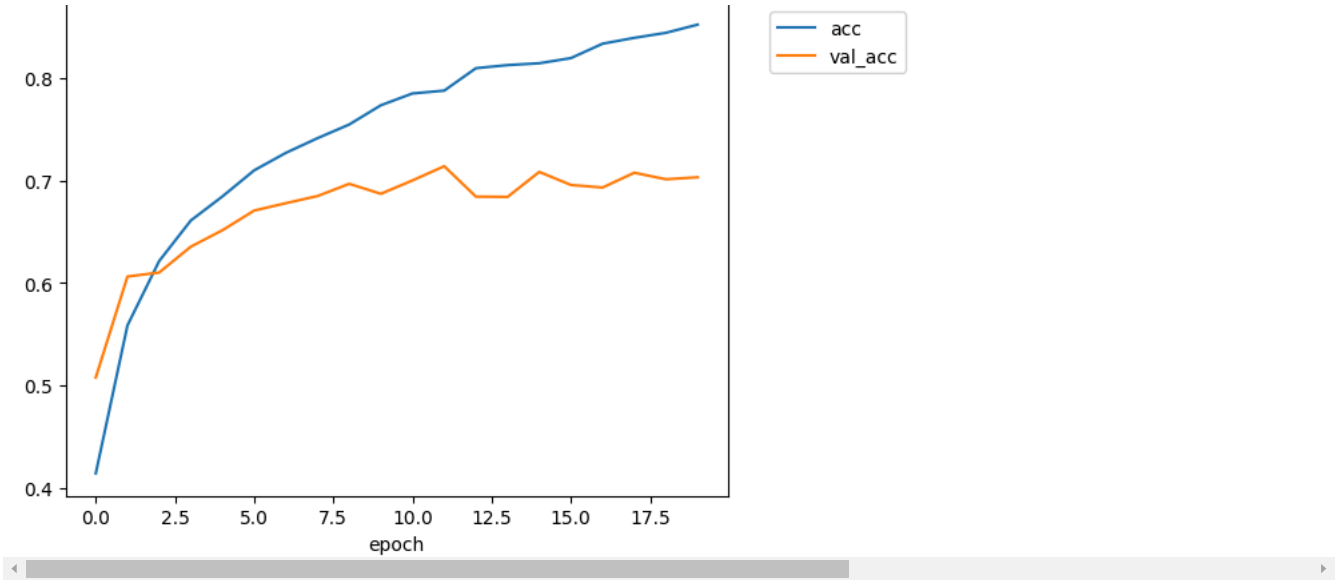
Model: "my_cnn_model"

Layer (type)	Output Shape	Param #
input_layer_13 (InputLayer)	(None, 32, 32, 3)	0
conv2d_6 (Conv2D)	(None, 30, 30, 32)	896
max_pooling2d_6 (MaxPooling2D)	(None, 15, 15, 32)	0
conv2d_7 (Conv2D)	(None, 13, 13, 64)	18,496
max_pooling2d_7 (MaxPooling2D)	(None, 6, 6, 64)	0
flatten_11 (Flatten)	(None, 2304)	0
dense_45 (Dense)	(None, 4)	9,220

Total params: 28,612 (111.77 KB)
Trainable params: 28,612 (111.77 KB)
Non-trainable params: 0 (0.00 B)

```
Epoch 1/20
168/168 ━━━━━━━━━━━━━━━━━━━━ 3s 8ms/step - accuracy: 0.3548 - loss: 8.2875 - val_accuracy: 0.5075 - val_loss: 1.2061
Epoch 2/20
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.5358 - loss: 1.1131 - val_accuracy: 0.6061 - val_loss: 0.9752
Epoch 3/20
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.6153 - loss: 0.9523 - val_accuracy: 0.6100 - val_loss: 0.9490
Epoch 4/20
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.6614 - loss: 0.8515 - val_accuracy: 0.6353 - val_loss: 0.9043
Epoch 5/20
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.6810 - loss: 0.8010 - val_accuracy: 0.6514 - val_loss: 0.9022
Epoch 6/20
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.7093 - loss: 0.7509 - val_accuracy: 0.6706 - val_loss: 0.8494
Epoch 7/20
168/168 ━━━━━━━━━━━━━━━━━━━━ 2s 5ms/step - accuracy: 0.7331 - loss: 0.6952 - val_accuracy: 0.6778 - val_loss: 0.8303
Epoch 8/20
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.7440 - loss: 0.6537 - val_accuracy: 0.6847 - val_loss: 0.8345
Epoch 9/20
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 5ms/step - accuracy: 0.7665 - loss: 0.6154 - val_accuracy: 0.6967 - val_loss: 0.8152
Epoch 10/20
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.7847 - loss: 0.5678 - val_accuracy: 0.6869 - val_loss: 0.8163
Epoch 11/20
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.7852 - loss: 0.5597 - val_accuracy: 0.7000 - val_loss: 0.8274
Epoch 12/20
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.7922 - loss: 0.5427 - val_accuracy: 0.7139 - val_loss: 0.8000
Epoch 13/20
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.8181 - loss: 0.4900 - val_accuracy: 0.6842 - val_loss: 0.9031
Epoch 14/20
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.8181 - loss: 0.4761 - val_accuracy: 0.6839 - val_loss: 0.9418
Epoch 15/20
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.8144 - loss: 0.4786 - val_accuracy: 0.7083 - val_loss: 0.8744
Epoch 16/20
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.8262 - loss: 0.4501 - val_accuracy: 0.6956 - val_loss: 0.9117
Epoch 17/20
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.8361 - loss: 0.4351 - val_accuracy: 0.6931 - val_loss: 0.9097
Epoch 18/20
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.8418 - loss: 0.4140 - val_accuracy: 0.7075 - val_loss: 0.8958
Epoch 19/20
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.8525 - loss: 0.3881 - val_accuracy: 0.7011 - val_loss: 0.9700
Epoch 20/20
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 4ms/step - accuracy: 0.8542 - loss: 0.3816 - val_accuracy: 0.7031 - val_loss: 1.0137
168/168 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.8662 - loss: 0.3551
36/36 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7129 - loss: 0.9700
Performance on the TRAIN set, ACCURACY= 0.8677380681037903
Performance on the TEST set, ACCURACY= 0.7036111354827881
```





[QUESTION] Fix your overfitting issue with a technique of your choice.

```
#Your code here

#To fight overfitting with CNN, we can use the exact same techniques of dropping out, regularization and batch normalization

#this time let's do data augmentation
from tensorflow.keras.preprocessing.image import ImageDataGenerator

#here we can see that data augmentation is by far the best method to fight overfitting, and at the same time it improves the performance

datagen = ImageDataGenerator(
    rotation_range=15,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True
)
datagen.fit(X_train)


# (1) DEFINE THE ARCHITECTURE OF MY MODEL
#first, I define all the layers and the way they are connected
inputs = tf.keras.Input(shape=(32,32,3)) #my input layer

x = tf.keras.layers.Conv2D(32,kernel_size=3,activation='relu')(inputs)
x = tf.keras.layers.MaxPooling2D()(x)
x = tf.keras.layers.Conv2D(64,kernel_size=3,activation='relu')(x)
x = tf.keras.layers.MaxPooling2D()(x)

x = tf.keras.layers.Flatten()(x) #cf. question below...

outputs = tf.keras.layers.Dense(4, activation='softmax')(x) # my output layer
#Then, I define my model with the input layer, the output layer and a name
my_cnn_model = tf.keras.Model(inputs=inputs, outputs=outputs, name="my_cnn_model")

#PRINT A SUMMARY OF THE ARCHITECTURE OF MY MODEL WITH THE NUMBER OF TRAINABLE PARAMETERS
my_cnn_model.summary()

# (2) DEFINE THE TRAINING HYPER-PARAMETERS WITH THE "COMPILE" METHOD:
'''
(1) Set the "optimizer" [pick 'adam', 'sgd' or 'rmsprop']
(2) Set the loss [cf. lesson #3, we pick the categorical cross-entropy]
(3) Set the final performance metric to evaluate the model
'''


my_cnn_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# (3) NOW, LET'S TRAIN ON MY DATA WITH THE "FIT" METHOD
'''
(1) Set the number of epochs
(2) Set the size of the (mini)batch
(3) Set the training dataset ==> here, X_train with Y_train
(4) Set the validation dataset (X_val, Y_val)
'''
nb_epochs=20
batch_size=100
training_history = my_cnn_model.fit(datagen.flow(X_train, Y_train, batch_size=batch_size),
```

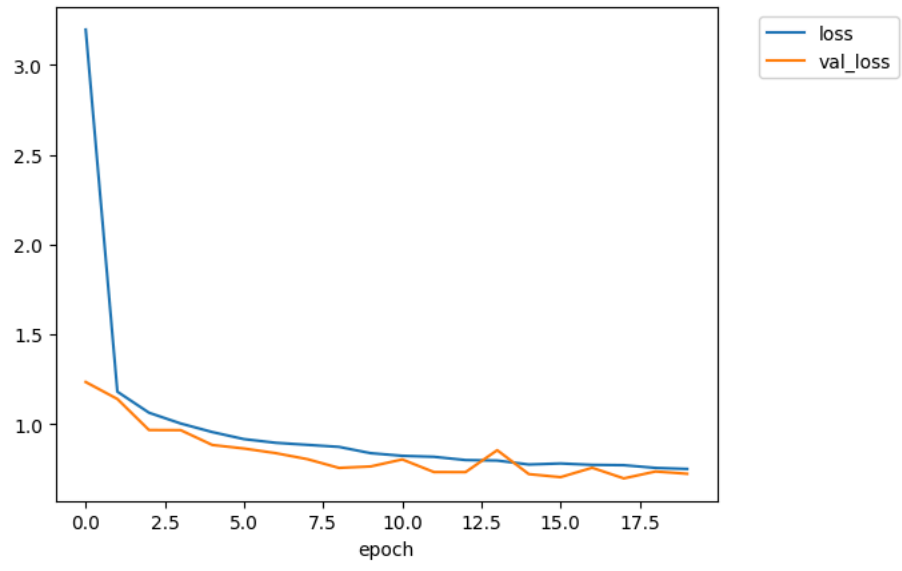
```
                                    validation_data=(X_val, Y_val),
                                    epochs=nb_epochs)

    #COMPUTE THE ACCURACY ON THE TRAINING AND TEST SETS
    loss_train, acc_train = my_cnn_model.evaluate(X_train, Y_train, batch_size=batch_size)
    loss_test, acc_test = my_cnn_model.evaluate(X_test, Y_test, batch_size=batch_size)

    print("Performance on the TRAIN set, ACCURACY=",acc_train)
    print("Performance on the TEST set, ACCURACY=",acc_test)

    display_training_curves(training_history)
```

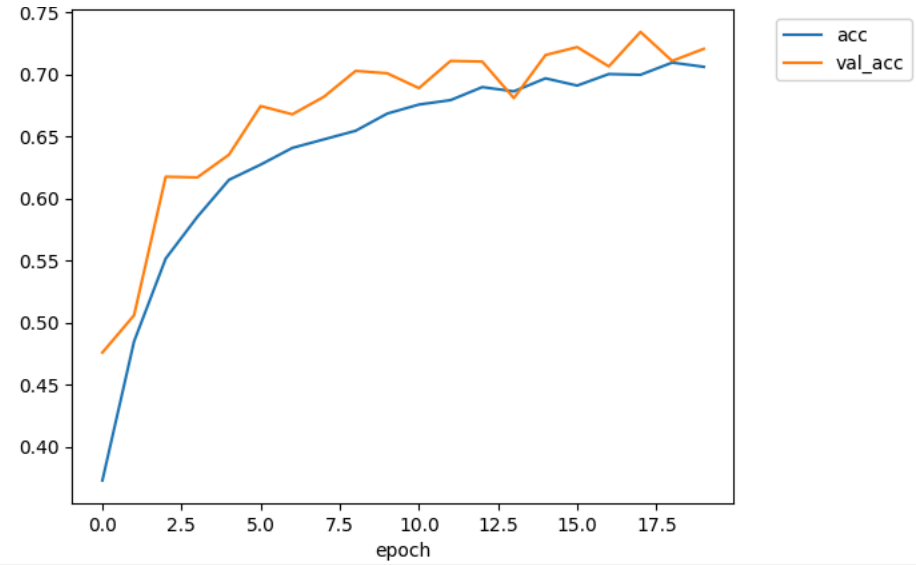
Model: "my_cnn_model"

Layer (type)	Output Shape	Param #
input_layer_3 (InputLayer)	(None, 32, 32, 3)	0
conv2d_2 (Conv2D)	(None, 30, 30, 32)	896
max_pooling2d_2 (MaxPooling2D)	(None, 15, 15, 32)	0
conv2d_3 (Conv2D)	(None, 13, 13, 64)	18,496
max_pooling2d_3 (MaxPooling2D)	(None, 6, 6, 64)	0
flatten_3 (Flatten)	(None, 2304)	0
dense_5 (Dense)	(None, 4)	9,220

Total params: 28,612 (111.77 KB)
Trainable params: 28,612 (111.77 KB)
Non-trainable params: 0 (0.00 B)

```
Epoch 1/20
168/168 ━━━━━━━━━━━━━━━━━━━━ 14s 71ms/step - accuracy: 0.3367 - loss: 7.4771 - val_accuracy: 0.4758 - val_loss: 1.2342
Epoch 2/20
168/168 ━━━━━━━━━━━━━━━━━━━━ 19s 67ms/step - accuracy: 0.4642 - loss: 1.2131 - val_accuracy: 0.5058 - val_loss: 1.1396
Epoch 3/20
168/168 ━━━━━━━━━━━━━━━━━━━━ 12s 71ms/step - accuracy: 0.5416 - loss: 1.0850 - val_accuracy: 0.6175 - val_loss: 0.9666
Epoch 4/20
168/168 ━━━━━━━━━━━━━━━━━━━━ 13s 73ms/step - accuracy: 0.5852 - loss: 1.0016 - val_accuracy: 0.6169 - val_loss: 0.9661
Epoch 5/20
168/168 ━━━━━━━━━━━━━━━━━━━━ 12s 71ms/step - accuracy: 0.6103 - loss: 0.9665 - val_accuracy: 0.6353 - val_loss: 0.8834
Epoch 6/20
168/168 ━━━━━━━━━━━━━━━━━━━━ 21s 73ms/step - accuracy: 0.6282 - loss: 0.9155 - val_accuracy: 0.6744 - val_loss: 0.8637
Epoch 7/20
168/168 ━━━━━━━━━━━━━━━━━━━━ 13s 74ms/step - accuracy: 0.6332 - loss: 0.9120 - val_accuracy: 0.6678 - val_loss: 0.8380
Epoch 8/20
168/168 ━━━━━━━━━━━━━━━━━━━━ 16s 96ms/step - accuracy: 0.6359 - loss: 0.9035 - val_accuracy: 0.6819 - val_loss: 0.8045
Epoch 9/20
168/168 ━━━━━━━━━━━━━━━━━━━━ 13s 77ms/step - accuracy: 0.6563 - loss: 0.8720 - val_accuracy: 0.7028 - val_loss: 0.7556
Epoch 10/20
168/168 ━━━━━━━━━━━━━━━━━━━━ 22s 87ms/step - accuracy: 0.6760 - loss: 0.8264 - val_accuracy: 0.7008 - val_loss: 0.7636
Epoch 11/20
168/168 ━━━━━━━━━━━━━━━━━━━━ 14s 80ms/step - accuracy: 0.6801 - loss: 0.8165 - val_accuracy: 0.6889 - val_loss: 0.8028
Epoch 12/20
168/168 ━━━━━━━━━━━━━━━━━━━━ 12s 67ms/step - accuracy: 0.6791 - loss: 0.8250 - val_accuracy: 0.7108 - val_loss: 0.7325
Epoch 13/20
168/168 ━━━━━━━━━━━━━━━━━━━━ 23s 81ms/step - accuracy: 0.6987 - loss: 0.7810 - val_accuracy: 0.7103 - val_loss: 0.7322
Epoch 14/20
168/168 ━━━━━━━━━━━━━━━━━━━━ 13s 75ms/step - accuracy: 0.6903 - loss: 0.7963 - val_accuracy: 0.6808 - val_loss: 0.8543
Epoch 15/20
168/168 ━━━━━━━━━━━━━━━━━━━━ 13s 73ms/step - accuracy: 0.7001 - loss: 0.7627 - val_accuracy: 0.7156 - val_loss: 0.7209
Epoch 16/20
168/168 ━━━━━━━━━━━━━━━━━━━━ 12s 71ms/step - accuracy: 0.6853 - loss: 0.7938 - val_accuracy: 0.7219 - val_loss: 0.7040
Epoch 17/20
168/168 ━━━━━━━━━━━━━━━━━━━━ 12s 70ms/step - accuracy: 0.7038 - loss: 0.7621 - val_accuracy: 0.7064 - val_loss: 0.7565
Epoch 18/20
168/168 ━━━━━━━━━━━━━━━━━━━━ 22s 79ms/step - accuracy: 0.6979 - loss: 0.7762 - val_accuracy: 0.7342 - val_loss: 0.6972
Epoch 19/20
168/168 ━━━━━━━━━━━━━━━━━━━━ 13s 75ms/step - accuracy: 0.7180 - loss: 0.7384 - val_accuracy: 0.7108 - val_loss: 0.7357
Epoch 20/20
168/168 ━━━━━━━━━━━━━━━━━━━━ 14s 84ms/step - accuracy: 0.7100 - loss: 0.7461 - val_accuracy: 0.7206 - val_loss: 0.7230
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 3ms/step - accuracy: 0.7230 - loss: 0.7153
36/36 ━━━━━━━━━━━━━━━━━━━━ 0s 3ms/step - accuracy: 0.7130 - loss: 0.7350
Performance on the TRAIN set, ACCURACY= 0.7289285659790039
Performance on the TEST set, ACCURACY= 0.7149999737739563
```





[**QUESTION**] Regarding these experiences, compare MLP vs. CNN

[**ANSWER**] In the CIFAR-4 classification task, **CNNs consistently outperform MLPs in terms of accuracy and parameter efficiency**. A deep MLP with 4.5M parameters achieved a test accuracy of 58% after 60 epochs, with significant overfitting, as the training accuracy reached 77%. In contrast, a simple CNN architecture with only 28K parameters achieved 71% test accuracy using data augmentation and ~70% without augmentation after 20 epochs. The training-test accuracy gap for CNNs was narrower (72% vs. 71%) compared to MLPs (77% vs. 58%), indicating better generalization. Data augmentation further improved CNN performance, reducing overfitting and increasing robustness, albeit with significantly longer training times (up to 18x slower per epoch). Thus, CNNs demonstrated superior performance, higher parameter efficiency, and better generalization on CIFAR-4 compared to MLPs.

BONUS

Double-cliquez (ou appuyez sur Entrée) pour modifier

```
#Your code here

#To fight overfitting with CNN, we can use the exact same techniques of dropping out, regularization and batch normalization

#this time let's do data augmentation
from tensorflow.keras.preprocessing.image import ImageDataGenerator

#here we can see that data augmentation is by far the best method to fight overfitting, and at the same time it improves the performance

datagen = ImageDataGenerator(
    rotation_range=15,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True
)
datagen.fit(X_train)


# (1) DEFINE THE ARCHITECTURE OF MY MODEL
#first, I define all the layers and the way they are connected
inputs = tf.keras.Input(shape=(32,32,3)) #my input layer

x = tf.keras.layers.Conv2D(32,kernel_size=3,activation='relu')(inputs)
x = tf.keras.layers.MaxPooling2D()(x)
x = tf.keras.layers.Conv2D(64,kernel_size=3,activation='relu')(x)
x = tf.keras.layers.MaxPooling2D()(x)

x = tf.keras.layers.Flatten()(x) #cf. question below...

outputs = tf.keras.layers.Dense(4, activation='softmax')(x) # my output layer
#Then, I define my model with the input layer, the output layer and a name
my_cnn_model = tf.keras.Model(inputs=inputs, outputs=outputs, name="my_cnn_model")

#PRINT A SUMMARY OF THE ARCHITECTURE OF MY MODEL WITH THE NUMBER OF TRAINABLE PARAMETERS
my_cnn_model.summary()

# (2) DEFINE THE TRAINING HYPER-PARAMETERS WITH THE "COMPILE" METHOD:
```

```
'''
(1) Set the "optimizer" [pick 'adam', 'sgd' or 'rmsprop']
(2) Set the loss [cf. lesson #3, we pick the categorical cross-entropy]
(3) Set the final performance metric to evaluate the model
'''

my_cnn_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# (3) NOW, LET'S TRAIN ON MY DATA WITH THE "FIT" METHOD
'''
(1) Set the number of epochs
(2) Set the size of the (mini)batch
(3) Set the training dataset ==> here, X_train with Y_train
(4) Set the validation dataset (X_val, Y_val)
'''
nb_epochs=60
batch_size=100
training_history = my_cnn_model.fit(datagen.flow(X_train, Y_train, batch_size=batch_size),
                                    validation_data=(X_val, Y_val),
                                    epochs=nb_epochs)

#COMPUTE THE ACCURACY ON THE TRAINING AND TEST SETS
loss_train, acc_train = my_cnn_model.evaluate(X_train, Y_train, batch_size=batch_size)
loss_test, acc_test = my_cnn_model.evaluate(X_test, Y_test, batch_size=batch_size)

print("Performance on the TRAIN set, ACCURACY=",acc_train)
print("Performance on the TEST set, ACCURACY=",acc_test)
```

Model: "my_cnn_model"

Layer (type)	Output Shape	Param #
input_layer_2 (InputLayer)	(None, 32, 32, 3)	0
conv2d (Conv2D)	(None, 30, 30, 32)	896
max_pooling2d (MaxPooling2D)	(None, 15, 15, 32)	0
conv2d_1 (Conv2D)	(None, 13, 13, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 6, 6, 64)	0
flatten_2 (Flatten)	(None, 2304)	0
dense_4 (Dense)	(None, 4)	9,220

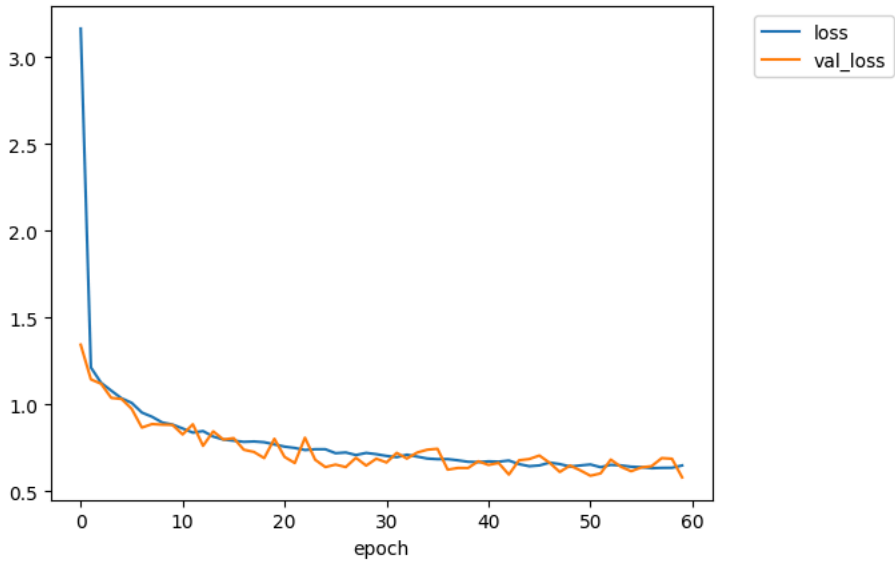
Total params: 28,612 (111.77 KB)
Trainable params: 28,612 (111.77 KB)
Non-trainable params: 0 (0.00 B)

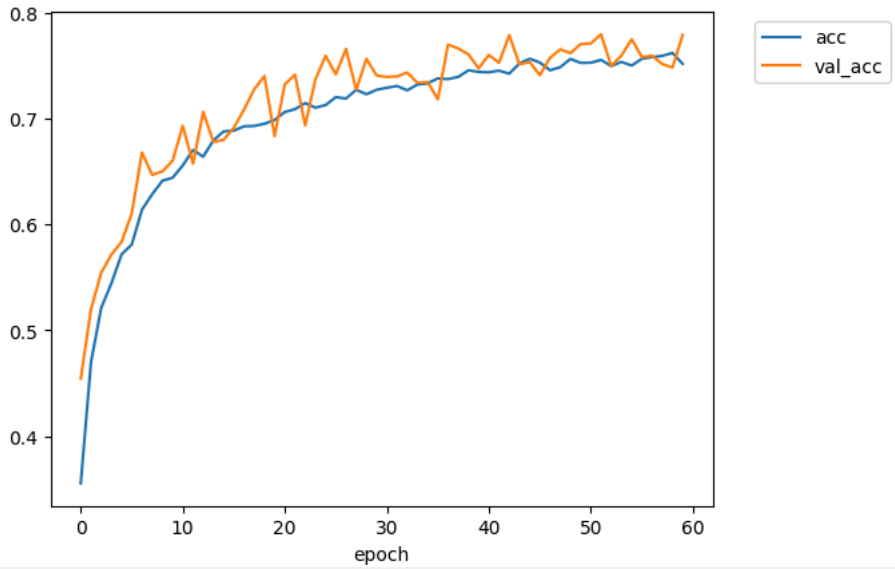
```
Epoch 1/60
/usr/local/lib/python3.10/dist-packages/keras/src/trainers/data_adapters/py_dataset_adapter.py:122: UserWarning: Your `PyDataset` cl
  self._warn_if_super_not_called()
168/168 ━━━━━━━━━━━━━━━━━━━━ 17s 79ms/step - accuracy: 0.3135 - loss: 7.2477 - val_accuracy: 0.4542 - val_loss: 1.3439
Epoch 2/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 16s 65ms/step - accuracy: 0.4473 - loss: 1.2496 - val_accuracy: 0.5197 - val_loss: 1.1446
Epoch 3/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 21s 70ms/step - accuracy: 0.5069 - loss: 1.1427 - val_accuracy: 0.5542 - val_loss: 1.1192
Epoch 4/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 20s 70ms/step - accuracy: 0.5410 - loss: 1.0860 - val_accuracy: 0.5714 - val_loss: 1.0379
Epoch 5/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 20s 69ms/step - accuracy: 0.5670 - loss: 1.0420 - val_accuracy: 0.5833 - val_loss: 1.0318
Epoch 6/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 12s 69ms/step - accuracy: 0.5789 - loss: 1.0128 - val_accuracy: 0.6103 - val_loss: 0.9751
Epoch 7/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 12s 69ms/step - accuracy: 0.6042 - loss: 0.9691 - val_accuracy: 0.6678 - val_loss: 0.8670
Epoch 8/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 22s 77ms/step - accuracy: 0.6244 - loss: 0.9343 - val_accuracy: 0.6467 - val_loss: 0.8871
Epoch 9/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 13s 72ms/step - accuracy: 0.6365 - loss: 0.9064 - val_accuracy: 0.6500 - val_loss: 0.8839
Epoch 10/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 20s 70ms/step - accuracy: 0.6504 - loss: 0.8814 - val_accuracy: 0.6600 - val_loss: 0.8826
Epoch 11/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 20s 67ms/step - accuracy: 0.6512 - loss: 0.8685 - val_accuracy: 0.6931 - val_loss: 0.8274
Epoch 12/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 21s 72ms/step - accuracy: 0.6738 - loss: 0.8271 - val_accuracy: 0.6572 - val_loss: 0.8860
Epoch 13/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 20s 69ms/step - accuracy: 0.6590 - loss: 0.8582 - val_accuracy: 0.7061 - val_loss: 0.7618
Epoch 14/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 12s 70ms/step - accuracy: 0.6788 - loss: 0.8134 - val_accuracy: 0.6778 - val_loss: 0.8451
Epoch 15/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 12s 69ms/step - accuracy: 0.6972 - loss: 0.7760 - val_accuracy: 0.6797 - val_loss: 0.7980
Epoch 16/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 20s 65ms/step - accuracy: 0.6911 - loss: 0.7926 - val_accuracy: 0.6911 - val_loss: 0.8060
Epoch 17/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 12s 66ms/step - accuracy: 0.6966 - loss: 0.7804 - val_accuracy: 0.7083 - val_loss: 0.7394
Epoch 18/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 12s 71ms/step - accuracy: 0.6972 - loss: 0.7753 - val_accuracy: 0.7275 - val_loss: 0.7265
Epoch 19/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 20s 70ms/step - accuracy: 0.6998 - loss: 0.7713 - val_accuracy: 0.7400 - val_loss: 0.6911
Epoch 20/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 11s 64ms/step - accuracy: 0.6930 - loss: 0.7775 - val_accuracy: 0.6833 - val_loss: 0.8025
Epoch 21/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 12s 66ms/step - accuracy: 0.7052 - loss: 0.7518 - val_accuracy: 0.7322 - val_loss: 0.6978
Epoch 22/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 12s 70ms/step - accuracy: 0.7137 - loss: 0.7361 - val_accuracy: 0.7414 - val_loss: 0.6628
Epoch 23/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 12s 69ms/step - accuracy: 0.7080 - loss: 0.7442 - val_accuracy: 0.6933 - val_loss: 0.8086
Epoch 24/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 20s 65ms/step - accuracy: 0.7072 - loss: 0.7450 - val_accuracy: 0.7369 - val_loss: 0.6816
Epoch 25/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 11s 62ms/step - accuracy: 0.7141 - loss: 0.7342 - val_accuracy: 0.7592 - val_loss: 0.6395
Epoch 26/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 12s 71ms/step - accuracy: 0.7213 - loss: 0.7090 - val_accuracy: 0.7417 - val_loss: 0.6538
Epoch 27/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 12s 70ms/step - accuracy: 0.7246 - loss: 0.7144 - val_accuracy: 0.7656 - val_loss: 0.6396
Epoch 28/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 20s 70ms/step - accuracy: 0.7249 - loss: 0.7150 - val_accuracy: 0.7269 - val_loss: 0.6930
Epoch 29/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 20s 70ms/step - accuracy: 0.7283 - loss: 0.7125 - val_accuracy: 0.7564 - val_loss: 0.6480
Epoch 30/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 22s 80ms/step - accuracy: 0.7238 - loss: 0.7201 - val_accuracy: 0.7406 - val_loss: 0.6879
Epoch 31/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 19s 70ms/step - accuracy: 0.7251 - loss: 0.7069 - val_accuracy: 0.7392 - val_loss: 0.6660
Epoch 32/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 20s 65ms/step - accuracy: 0.7336 - loss: 0.6938 - val_accuracy: 0.7397 - val_loss: 0.7212
Epoch 33/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 12s 67ms/step - accuracy: 0.7238 - loss: 0.7134 - val_accuracy: 0.7433 - val_loss: 0.6885
Epoch 34/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 12s 72ms/step - accuracy: 0.7350 - loss: 0.6911 - val_accuracy: 0.7336 - val_loss: 0.7230
```

```
168/168 ━━━━━━━━━━━━━━━━━━━━ 12s 72ms/step - accuracy: 0.7390 - loss: 0.6911 - val_accuracy: 0.7390 - val_loss: 0.7250
Epoch 35/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 12s 70ms/step - accuracy: 0.7281 - loss: 0.6987 - val_accuracy: 0.7344 - val_loss: 0.7395
Epoch 36/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 20s 69ms/step - accuracy: 0.7367 - loss: 0.6775 - val_accuracy: 0.7181 - val_loss: 0.7447
Epoch 37/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 21s 69ms/step - accuracy: 0.7363 - loss: 0.6894 - val_accuracy: 0.7697 - val_loss: 0.6247
Epoch 38/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 12s 69ms/step - accuracy: 0.7455 - loss: 0.6640 - val_accuracy: 0.7661 - val_loss: 0.6344
Epoch 39/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 22s 78ms/step - accuracy: 0.7423 - loss: 0.6678 - val_accuracy: 0.7606 - val_loss: 0.6341
Epoch 40/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 11s 63ms/step - accuracy: 0.7535 - loss: 0.6560 - val_accuracy: 0.7472 - val_loss: 0.6731
Epoch 41/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 18s 106ms/step - accuracy: 0.7465 - loss: 0.6676 - val_accuracy: 0.7600 - val_loss: 0.6520
Epoch 42/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 13s 64ms/step - accuracy: 0.7423 - loss: 0.6751 - val_accuracy: 0.7525 - val_loss: 0.6632
Epoch 43/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 11s 61ms/step - accuracy: 0.7403 - loss: 0.6807 - val_accuracy: 0.7786 - val_loss: 0.5962
Epoch 44/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 12s 69ms/step - accuracy: 0.7549 - loss: 0.6478 - val_accuracy: 0.7511 - val_loss: 0.6787
Epoch 45/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 12s 68ms/step - accuracy: 0.7531 - loss: 0.6516 - val_accuracy: 0.7533 - val_loss: 0.6858
Epoch 46/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 20s 67ms/step - accuracy: 0.7554 - loss: 0.6409 - val_accuracy: 0.7408 - val_loss: 0.7059
Epoch 47/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 21s 70ms/step - accuracy: 0.7423 - loss: 0.6687 - val_accuracy: 0.7572 - val_loss: 0.6640
Epoch 48/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 12s 68ms/step - accuracy: 0.7542 - loss: 0.6424 - val_accuracy: 0.7653 - val_loss: 0.6113
Epoch 49/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 12s 70ms/step - accuracy: 0.7645 - loss: 0.6246 - val_accuracy: 0.7617 - val_loss: 0.6484
Epoch 50/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 12s 69ms/step - accuracy: 0.7508 - loss: 0.6543 - val_accuracy: 0.7703 - val_loss: 0.6208
Epoch 51/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 11s 61ms/step - accuracy: 0.7575 - loss: 0.6506 - val_accuracy: 0.7708 - val_loss: 0.5898
Epoch 52/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 12s 66ms/step - accuracy: 0.7579 - loss: 0.6360 - val_accuracy: 0.7794 - val_loss: 0.6029
Epoch 53/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 21s 69ms/step - accuracy: 0.7551 - loss: 0.6446 - val_accuracy: 0.7494 - val_loss: 0.6826
Epoch 54/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 27s 100ms/step - accuracy: 0.7494 - loss: 0.6531 - val_accuracy: 0.7594 - val_loss: 0.6410
Epoch 55/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 13s 73ms/step - accuracy: 0.7438 - loss: 0.6568 - val_accuracy: 0.7747 - val_loss: 0.6157
Epoch 56/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 16s 92ms/step - accuracy: 0.7693 - loss: 0.6098 - val_accuracy: 0.7581 - val_loss: 0.6375
Epoch 57/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 16s 91ms/step - accuracy: 0.7542 - loss: 0.6430 - val_accuracy: 0.7592 - val_loss: 0.6446
Epoch 58/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 13s 73ms/step - accuracy: 0.7572 - loss: 0.6354 - val_accuracy: 0.7514 - val_loss: 0.6911
Epoch 59/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 11s 60ms/step - accuracy: 0.7621 - loss: 0.6377 - val_accuracy: 0.7481 - val_loss: 0.6875
Epoch 60/60
168/168 ━━━━━━━━━━━━━━━━━━━━ 12s 71ms/step - accuracy: 0.7512 - loss: 0.6464 - val_accuracy: 0.7789 - val_loss: 0.5801
168/168 ━━━━━━━━━━━━━━━━━━━━ 1s 2ms/step - accuracy: 0.8020 - loss: 0.5307
36/36 ━━━━━━━━━━━━━━━━━━━━ 0s 2ms/step - accuracy: 0.7853 - loss: 0.5974
Performance on the TRAIN set, ACCURACY= 0.8045833110809326
Performance on the TEST set, ACCURACY= 0.7827777862548828
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-12-ed0303bc2989> in <cell line: 66>()
     64 print("Performance on the TEST set, ACCURACY=",acc_test)
     65
---> 66 display_training_curves(training_history)
     67

NameError: name 'display_training_curves' is not defined
```

Étapes suivantes : Expliquer l'erreur

```
display_training_curves(training_history)
```





[ANSWER] This CNN achieved ~78.3% test accuracy, with data augmentation **and 60 epochs** eliminating completely the gap generalization for 60 epochs or less, albeit at 18x slower training time. This simple architecture (28,612 parameters) effectively captured spatial patterns, proving CNNs' strength in image classification by an outstanding performance of 78%. Which is the best result we get trough all of our different MLP algorithms.

```
#Your code here

#Here we will do some transfer learning with the ResNet50 algorithm, a pre-trained deep convolutional neural network, for our CIFAR-4 da

from tensorflow.keras.applications import ResNet50
from tensorflow.keras.layers import Dense, Flatten, GlobalAveragePooling2D, Input

# (1) DEFINE THE ARCHITECTURE OF MY MODEL
#first, I define all the layers and the way they are connected

# Load the ResNet50 model pre-trained on ImageNet
base_model = ResNet50(weights='imagenet', include_top=False, input_shape=(32, 32, 3))

# Freeze the base model (optional if you only want to train the top layers)
base_model.trainable = False

# Add custom layers for CIFAR-4 classification
inputs = base_model.input
x = base_model.output
x = GlobalAveragePooling2D()(x)  # Pooling to reduce dimensions
x = Dense(256, activation='relu')(x)  # Fully connected layer
```

```
x = Dense(128, activation='relu')(x)  # Fully connected layer

outputs = tf.keras.layers.Dense(4, activation='softmax')(x) # my output layer
#Then, I define my model with the input layer, the output layer and a name
my_res50_model = tf.keras.Model(inputs=inputs, outputs=outputs, name="my_res50_model")

#PRINT A SUMMARY OF THE ARCHITECTURE OF MY MODEL WITH THE NUMBER OF TRAINABLE PARAMETERS
my_res50_model.summary()

# (2) DEFINE THE TRAINING HYPER-PARAMETERS WITH THE "COMPILE" METHOD:
'''
(1) Set the "optimizer" [pick 'adam', 'sgd' or 'rmsprop']
(2) Set the loss [cf. lesson #3, we pick the categorical cross-entropy]
(3) Set the final performance metric to evaluate the model
'''

my_res50_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# (3) NOW, LET'S TRAIN ON MY DATA WITH THE "FIT" METHOD
'''
(1) Set the number of epochs
(2) Set the size of the (mini)batch
(3) Set the training dataset ==> here, X_train with Y_train
(4) Set the validation dataset (X_val, Y_val)
'''
nb_epochs=15
batch_size=100
training_history = my_res50_model.fit(datagen.flow(X_train, Y_train, batch_size=batch_size),
                                      validation_data=(X_val, Y_val),
                                      epochs=nb_epochs,
                                      batch_size=batch_size)

#COMPUTE THE ACCURACY ON THE TRAINING AND TEST SETS
loss_train, acc_train = my_res50_model.evaluate(X_train, Y_train, batch_size=batch_size)
loss_test, acc_test = my_res50_model.evaluate(X_test, Y_test, batch_size=batch_size)

print("Performance on the TRAIN set, ACCURACY=",acc_train)
print("Performance on the TEST set, ACCURACY=",acc_test)

display_training_curves(training_history)
```

Model: "my_res50_model"

Layer (type)	Output Shape	Param #	Connected to
input_layer_14 (InputLayer)	(None, 32, 32, 3)	0	-
conv1_pad (ZeroPadding2D)	(None, 38, 38, 3)	0	input_layer_14[0][0]
conv1_conv (Conv2D)	(None, 16, 16, 64)	9,472	conv1_pad[0][0]
conv1_bn (BatchNormalization)	(None, 16, 16, 64)	256	conv1_conv[0][0]
conv1_relu (Activation)	(None, 16, 16, 64)	0	conv1_bn[0][0]
pool1_pad (ZeroPadding2D)	(None, 18, 18, 64)	0	conv1_relu[0][0]
pool1_pool (MaxPooling2D)	(None, 8, 8, 64)	0	pool1_pad[0][0]
conv2_block1_1_conv (Conv2D)	(None, 8, 8, 64)	4,160	pool1_pool[0][0]
conv2_block1_1_bn (BatchNormalization)	(None, 8, 8, 64)	256	conv2_block1_1_conv[0…
conv2_block1_1_relu (Activation)	(None, 8, 8, 64)	0	conv2_block1_1_bn[0][…
conv2_block1_2_conv (Conv2D)	(None, 8, 8, 64)	36,928	conv2_block1_1_relu[0…
conv2_block1_2_bn (BatchNormalization)	(None, 8, 8, 64)	256	conv2_block1_2_conv[0…
conv2_block1_2_relu (Activation)	(None, 8, 8, 64)	0	conv2_block1_2_bn[0][…
conv2_block1_0_conv (Conv2D)	(None, 8, 8, 256)	16,640	pool1_pool[0][0]
conv2_block1_3_conv (Conv2D)	(None, 8, 8, 256)	16,640	conv2_block1_2_relu[0…
conv2_block1_0_bn (BatchNormalization)	(None, 8, 8, 256)	1,024	conv2_block1_0_conv[0…
conv2_block1_3_bn (BatchNormalization)	(None, 8, 8, 256)	1,024	conv2_block1_3_conv[0…
conv2_block1_add (Add)	(None, 8, 8, 256)	0	conv2_block1_0_bn[0][… conv2_block1_3_bn[0][…
conv2_block1_out (Activation)	(None, 8, 8, 256)	0	conv2_block1_add[0][0]
conv2_block2_1_conv (Conv2D)	(None, 8, 8, 64)	16,448	conv2_block1_out[0][0]
conv2_block2_1_bn (BatchNormalization)	(None, 8, 8, 64)	256	conv2_block2_1_conv[0…
conv2_block2_1_relu (Activation)	(None, 8, 8, 64)	0	conv2_block2_1_bn[0][…
conv2_block2_2_conv (Conv2D)	(None, 8, 8, 64)	36,928	conv2_block2_1_relu[0…
conv2_block2_2_bn (BatchNormalization)	(None, 8, 8, 64)	256	conv2_block2_2_conv[0…
conv2_block2_2_relu (Activation)	(None, 8, 8, 64)	0	conv2_block2_2_bn[0][…
conv2_block2_3_conv (Conv2D)	(None, 8, 8, 256)	16,640	conv2_block2_2_relu[0…
conv2_block2_3_bn (BatchNormalization)	(None, 8, 8, 256)	1,024	conv2_block2_3_conv[0…
conv2_block2_add (Add)	(None, 8, 8, 256)	0	conv2_block1_out[0][0… conv2_block2_3_bn[0][…
conv2_block2_out (Activation)	(None, 8, 8, 256)	0	conv2_block2_add[0][0]
conv2_block3_1_conv (Conv2D)	(None, 8, 8, 64)	16,448	conv2_block2_out[0][0]
conv2_block3_1_bn (BatchNormalization)	(None, 8, 8, 64)	256	conv2_block3_1_conv[0…

conv2_block3_1_relu (Activation)	(None, 8, 8, 64)	0	conv2_block3_1_bn[0][…
conv2_block3_2_conv (Conv2D)	(None, 8, 8, 64)	36,928	conv2_block3_1_relu[0…
conv2_block3_2_bn (BatchNormalization)	(None, 8, 8, 64)	256	conv2_block3_2_conv[0…
conv2_block3_2_relu (Activation)	(None, 8, 8, 64)	0	conv2_block3_2_bn[0][…
conv2_block3_3_conv (Conv2D)	(None, 8, 8, 256)	16,640	conv2_block3_2_relu[0…
conv2_block3_3_bn (BatchNormalization)	(None, 8, 8, 256)	1,024	conv2_block3_3_conv[0…
conv2_block3_add (Add)	(None, 8, 8, 256)	0	conv2_block2_out[0][0… conv2_block3_3_bn[0][…
conv2_block3_out (Activation)	(None, 8, 8, 256)	0	conv2_block3_add[0][0]
conv3_block1_1_conv (Conv2D)	(None, 4, 4, 128)	32,896	conv2_block3_out[0][0]
conv3_block1_1_bn (BatchNormalization)	(None, 4, 4, 128)	512	conv3_block1_1_conv[0…
conv3_block1_1_relu (Activation)	(None, 4, 4, 128)	0	conv3_block1_1_bn[0][…
conv3_block1_2_conv (Conv2D)	(None, 4, 4, 128)	147,584	conv3_block1_1_relu[0…
conv3_block1_2_bn (BatchNormalization)	(None, 4, 4, 128)	512	conv3_block1_2_conv[0…
conv3_block1_2_relu (Activation)	(None, 4, 4, 128)	0	conv3_block1_2_bn[0][…
conv3_block1_0_conv (Conv2D)	(None, 4, 4, 512)	131,584	conv2_block3_out[0][0]
conv3_block1_3_conv (Conv2D)	(None, 4, 4, 512)	66,048	conv3_block1_2_relu[0…
conv3_block1_0_bn (BatchNormalization)	(None, 4, 4, 512)	2,048	conv3_block1_0_conv[0…
conv3_block1_3_bn (BatchNormalization)	(None, 4, 4, 512)	2,048	conv3_block1_3_conv[0…
conv3_block1_add (Add)	(None, 4, 4, 512)	0	conv3_block1_0_bn[0][… conv3_block1_3_bn[0][…
conv3_block1_out (Activation)	(None, 4, 4, 512)	0	conv3_block1_add[0][0]
conv3_block2_1_conv (Conv2D)	(None, 4, 4, 128)	65,664	conv3_block1_out[0][0]
conv3_block2_1_bn (BatchNormalization)	(None, 4, 4, 128)	512	conv3_block2_1_conv[0…
conv3_block2_1_relu (Activation)	(None, 4, 4, 128)	0	conv3_block2_1_bn[0][…
conv3_block2_2_conv (Conv2D)	(None, 4, 4, 128)	147,584	conv3_block2_1_relu[0…
conv3_block2_2_bn (BatchNormalization)	(None, 4, 4, 128)	512	conv3_block2_2_conv[0…
conv3_block2_2_relu (Activation)	(None, 4, 4, 128)	0	conv3_block2_2_bn[0][…
conv3_block2_3_conv (Conv2D)	(None, 4, 4, 512)	66,048	conv3_block2_2_relu[0…
conv3_block2_3_bn (BatchNormalization)	(None, 4, 4, 512)	2,048	conv3_block2_3_conv[0…
conv3_block2_add (Add)	(None, 4, 4, 512)	0	conv3_block1_out[0][0… conv3_block2_3_bn[0][…
conv3_block2_out (Activation)	(None, 4, 4, 512)	0	conv3_block2_add[0][0]
conv3_block3_1_conv (Conv2D)	(None, 4, 4, 128)	65,664	conv3_block2_out[0][0]

conv3_block3_1_bn (BatchNormalization)	(None, 4, 4, 128)	512	conv3_block3_1_conv[0…
conv3_block3_1_relu (Activation)	(None, 4, 4, 128)	0	conv3_block3_1_bn[0][…
conv3_block3_2_conv (Conv2D)	(None, 4, 4, 128)	147,584	conv3_block3_1_relu[0…
conv3_block3_2_bn (BatchNormalization)	(None, 4, 4, 128)	512	conv3_block3_2_conv[0…
conv3_block3_2_relu (Activation)	(None, 4, 4, 128)	0	conv3_block3_2_bn[0][…
conv3_block3_3_conv (Conv2D)	(None, 4, 4, 512)	66,048	conv3_block3_2_relu[0…
conv3_block3_3_bn (BatchNormalization)	(None, 4, 4, 512)	2,048	conv3_block3_3_conv[0…
conv3_block3_add (Add)	(None, 4, 4, 512)	0	conv3_block2_out[0][0… conv3_block3_3_bn[0][…
conv3_block3_out (Activation)	(None, 4, 4, 512)	0	conv3_block3_add[0][0]
conv3_block4_1_conv (Conv2D)	(None, 4, 4, 128)	65,664	conv3_block3_out[0][0]
conv3_block4_1_bn (BatchNormalization)	(None, 4, 4, 128)	512	conv3_block4_1_conv[0…
conv3_block4_1_relu (Activation)	(None, 4, 4, 128)	0	conv3_block4_1_bn[0][…
conv3_block4_2_conv (Conv2D)	(None, 4, 4, 128)	147,584	conv3_block4_1_relu[0…
conv3_block4_2_bn (BatchNormalization)	(None, 4, 4, 128)	512	conv3_block4_2_conv[0…
conv3_block4_2_relu (Activation)	(None, 4, 4, 128)	0	conv3_block4_2_bn[0][…
conv3_block4_3_conv (Conv2D)	(None, 4, 4, 512)	66,048	conv3_block4_2_relu[0…
conv3_block4_3_bn (BatchNormalization)	(None, 4, 4, 512)	2,048	conv3_block4_3_conv[0…
conv3_block4_add (Add)	(None, 4, 4, 512)	0	conv3_block3_out[0][0… conv3_block4_3_bn[0][…
conv3_block4_out (Activation)	(None, 4, 4, 512)	0	conv3_block4_add[0][0]
conv4_block1_1_conv (Conv2D)	(None, 2, 2, 256)	131,328	conv3_block4_out[0][0]
conv4_block1_1_bn (BatchNormalization)	(None, 2, 2, 256)	1,024	conv4_block1_1_conv[0…
conv4_block1_1_relu (Activation)	(None, 2, 2, 256)	0	conv4_block1_1_bn[0][…
conv4_block1_2_conv (Conv2D)	(None, 2, 2, 256)	590,080	conv4_block1_1_relu[0…
conv4_block1_2_bn (BatchNormalization)	(None, 2, 2, 256)	1,024	conv4_block1_2_conv[0…
conv4_block1_2_relu (Activation)	(None, 2, 2, 256)	0	conv4_block1_2_bn[0][…
conv4_block1_0_conv (Conv2D)	(None, 2, 2, 1024)	525,312	conv3_block4_out[0][0]
conv4_block1_3_conv (Conv2D)	(None, 2, 2, 1024)	263,168	conv4_block1_2_relu[0…
conv4_block1_0_bn (BatchNormalization)	(None, 2, 2, 1024)	4,096	conv4_block1_0_conv[0…
conv4_block1_3_bn (BatchNormalization)	(None, 2, 2, 1024)	4,096	conv4_block1_3_conv[0…
conv4_block1_add (Add)	(None, 2, 2, 1024)	0	conv4_block1_0_bn[0][… conv4_block1_3_bn[0][…
conv4_block1_out (Activation)	(None, 2, 2, 1024)	0	conv4_block1_add[0][0]

conv4_block2_1_conv (Conv2D)	(None, 2, 2, 256)	262,400	conv4_block1_out[0][0]
conv4_block2_1_bn (BatchNormalization)	(None, 2, 2, 256)	1,024	conv4_block2_1_conv[0…
conv4_block2_1_relu (Activation)	(None, 2, 2, 256)	0	conv4_block2_1_bn[0][…
conv4_block2_2_conv (Conv2D)	(None, 2, 2, 256)	590,080	conv4_block2_1_relu[0…
conv4_block2_2_bn (BatchNormalization)	(None, 2, 2, 256)	1,024	conv4_block2_2_conv[0…
conv4_block2_2_relu (Activation)	(None, 2, 2, 256)	0	conv4_block2_2_bn[0][…
conv4_block2_3_conv (Conv2D)	(None, 2, 2, 1024)	263,168	conv4_block2_2_relu[0…
conv4_block2_3_bn (BatchNormalization)	(None, 2, 2, 1024)	4,096	conv4_block2_3_conv[0…
conv4_block2_add (Add)	(None, 2, 2, 1024)	0	conv4_block1_out[0][0… conv4_block2_3_bn[0][…
conv4_block2_out (Activation)	(None, 2, 2, 1024)	0	conv4_block2_add[0][0]
conv4_block3_1_conv (Conv2D)	(None, 2, 2, 256)	262,400	conv4_block2_out[0][0]
conv4_block3_1_bn (BatchNormalization)	(None, 2, 2, 256)	1,024	conv4_block3_1_conv[0…
conv4_block3_1_relu (Activation)	(None, 2, 2, 256)	0	conv4_block3_1_bn[0][…
conv4_block3_2_conv (Conv2D)	(None, 2, 2, 256)	590,080	conv4_block3_1_relu[0…
conv4_block3_2_bn (BatchNormalization)	(None, 2, 2, 256)	1,024	conv4_block3_2_conv[0…
conv4_block3_2_relu (Activation)	(None, 2, 2, 256)	0	conv4_block3_2_bn[0][…
conv4_block3_3_conv (Conv2D)	(None, 2, 2, 1024)	263,168	conv4_block3_2_relu[0…
conv4_block3_3_bn (BatchNormalization)	(None, 2, 2, 1024)	4,096	conv4_block3_3_conv[0…
conv4_block3_add (Add)	(None, 2, 2, 1024)	0	conv4_block2_out[0][0… conv4_block3_3_bn[0][…
conv4_block3_out (Activation)	(None, 2, 2, 1024)	0	conv4_block3_add[0][0]
conv4_block4_1_conv (Conv2D)	(None, 2, 2, 256)	262,400	conv4_block3_out[0][0]
conv4_block4_1_bn (BatchNormalization)	(None, 2, 2, 256)	1,024	conv4_block4_1_conv[0…
conv4_block4_1_relu (Activation)	(None, 2, 2, 256)	0	conv4_block4_1_bn[0][…
conv4_block4_2_conv (Conv2D)	(None, 2, 2, 256)	590,080	conv4_block4_1_relu[0…
conv4_block4_2_bn (BatchNormalization)	(None, 2, 2, 256)	1,024	conv4_block4_2_conv[0…
conv4_block4_2_relu (Activation)	(None, 2, 2, 256)	0	conv4_block4_2_bn[0][…
conv4_block4_3_conv (Conv2D)	(None, 2, 2, 1024)	263,168	conv4_block4_2_relu[0…
conv4_block4_3_bn (BatchNormalization)	(None, 2, 2, 1024)	4,096	conv4_block4_3_conv[0…
conv4_block4_add (Add)	(None, 2, 2, 1024)	0	conv4_block3_out[0][0… conv4_block4_3_bn[0][…
conv4_block4_out (Activation)	(None, 2, 2, 1024)	0	conv4_block4_add[0][0]
conv4_block5_1_conv	(None, 2, 2, 256)	262,400	conv4_block4_out[0][0]

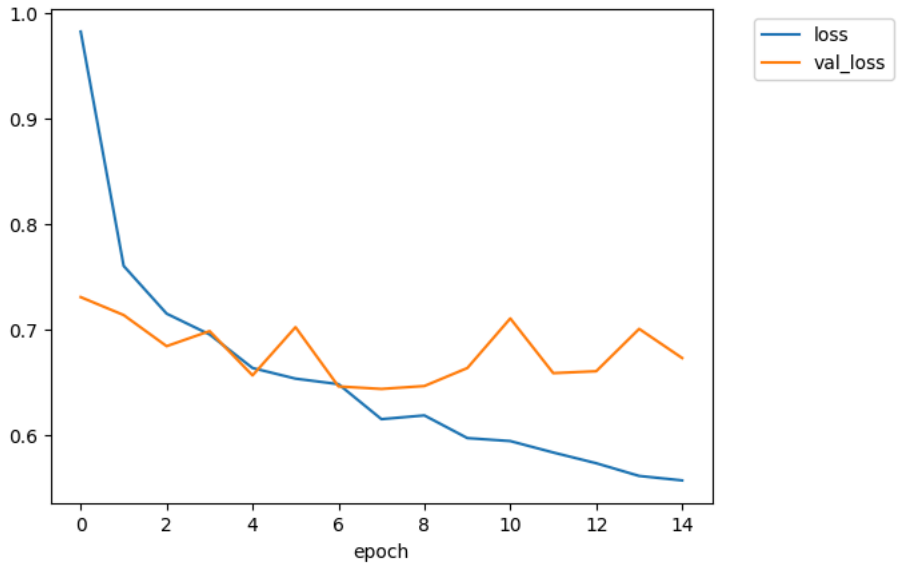
(Conv2D)			
conv4_block5_1_bn (BatchNormalization)	(None, 2, 2, 256)	1,024	conv4_block5_1_conv[0…
conv4_block5_1_relu (Activation)	(None, 2, 2, 256)	0	conv4_block5_1_bn[0][…
conv4_block5_2_conv (Conv2D)	(None, 2, 2, 256)	590,080	conv4_block5_1_relu[0…
conv4_block5_2_bn (BatchNormalization)	(None, 2, 2, 256)	1,024	conv4_block5_2_conv[0…
conv4_block5_2_relu (Activation)	(None, 2, 2, 256)	0	conv4_block5_2_bn[0][…
conv4_block5_3_conv (Conv2D)	(None, 2, 2, 1024)	263,168	conv4_block5_2_relu[0…
conv4_block5_3_bn (BatchNormalization)	(None, 2, 2, 1024)	4,096	conv4_block5_3_conv[0…
conv4_block5_add (Add)	(None, 2, 2, 1024)	0	conv4_block4_out[0][0… conv4_block5_3_bn[0][…
conv4_block5_out (Activation)	(None, 2, 2, 1024)	0	conv4_block5_add[0][0]
conv4_block6_1_conv (Conv2D)	(None, 2, 2, 256)	262,400	conv4_block5_out[0][0]
conv4_block6_1_bn (BatchNormalization)	(None, 2, 2, 256)	1,024	conv4_block6_1_conv[0…
conv4_block6_1_relu (Activation)	(None, 2, 2, 256)	0	conv4_block6_1_bn[0][…
conv4_block6_2_conv (Conv2D)	(None, 2, 2, 256)	590,080	conv4_block6_1_relu[0…
conv4_block6_2_bn (BatchNormalization)	(None, 2, 2, 256)	1,024	conv4_block6_2_conv[0…
conv4_block6_2_relu (Activation)	(None, 2, 2, 256)	0	conv4_block6_2_bn[0][…
conv4_block6_3_conv (Conv2D)	(None, 2, 2, 1024)	263,168	conv4_block6_2_relu[0…
conv4_block6_3_bn (BatchNormalization)	(None, 2, 2, 1024)	4,096	conv4_block6_3_conv[0…
conv4_block6_add (Add)	(None, 2, 2, 1024)	0	conv4_block5_out[0][0… conv4_block6_3_bn[0][…
conv4_block6_out (Activation)	(None, 2, 2, 1024)	0	conv4_block6_add[0][0]
conv5_block1_1_conv (Conv2D)	(None, 1, 1, 512)	524,800	conv4_block6_out[0][0]
conv5_block1_1_bn (BatchNormalization)	(None, 1, 1, 512)	2,048	conv5_block1_1_conv[0…
conv5_block1_1_relu (Activation)	(None, 1, 1, 512)	0	conv5_block1_1_bn[0][…
conv5_block1_2_conv (Conv2D)	(None, 1, 1, 512)	2,359,808	conv5_block1_1_relu[0…
conv5_block1_2_bn (BatchNormalization)	(None, 1, 1, 512)	2,048	conv5_block1_2_conv[0…
conv5_block1_2_relu (Activation)	(None, 1, 1, 512)	0	conv5_block1_2_bn[0][…
conv5_block1_0_conv (Conv2D)	(None, 1, 1, 2048)	2,099,200	conv4_block6_out[0][0]
conv5_block1_3_conv (Conv2D)	(None, 1, 1, 2048)	1,050,624	conv5_block1_2_relu[0…
conv5_block1_0_bn (BatchNormalization)	(None, 1, 1, 2048)	8,192	conv5_block1_0_conv[0…
conv5_block1_3_bn (BatchNormalization)	(None, 1, 1, 2048)	8,192	conv5_block1_3_conv[0…
conv5_block1_add (Add)	(None, 1, 1, 2048)	0	conv5_block1_0_bn[0][… conv5_block1_3_bn[0][…
conv5_block1_out	(None, 1, 1, 2048)	0	conv5_block1_add[0][0]

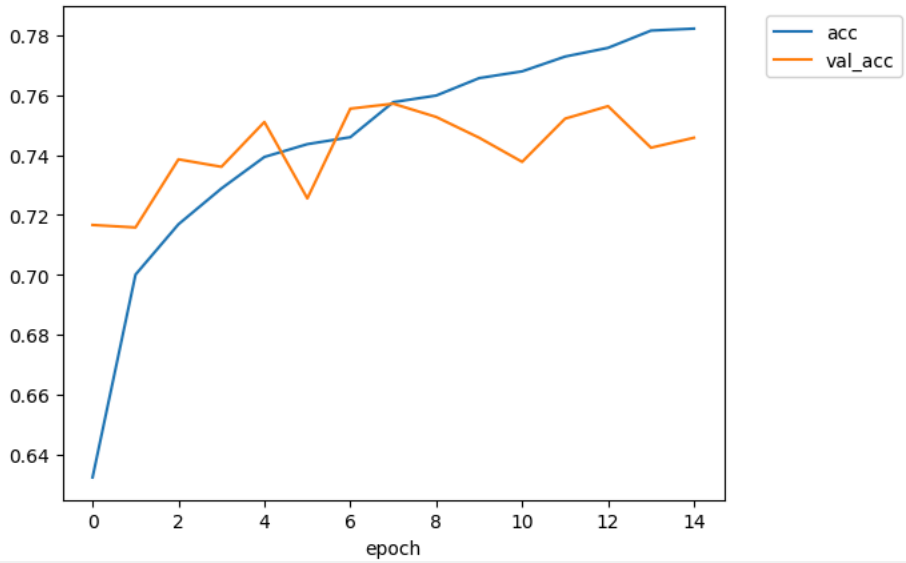
(Activation)			
conv5_block2_1_conv (Conv2D)	(None, 1, 1, 512)	1,049,088	conv5_block1_out[0][0]
conv5_block2_1_bn (BatchNormalization)	(None, 1, 1, 512)	2,048	conv5_block2_1_conv[0…
conv5_block2_1_relu (Activation)	(None, 1, 1, 512)	0	conv5_block2_1_bn[0][…
conv5_block2_2_conv (Conv2D)	(None, 1, 1, 512)	2,359,808	conv5_block2_1_relu[0…
conv5_block2_2_bn (BatchNormalization)	(None, 1, 1, 512)	2,048	conv5_block2_2_conv[0…
conv5_block2_2_relu (Activation)	(None, 1, 1, 512)	0	conv5_block2_2_bn[0][…
conv5_block2_3_conv (Conv2D)	(None, 1, 1, 2048)	1,050,624	conv5_block2_2_relu[0…
conv5_block2_3_bn (BatchNormalization)	(None, 1, 1, 2048)	8,192	conv5_block2_3_conv[0…
conv5_block2_add (Add)	(None, 1, 1, 2048)	0	conv5_block1_out[0][0… conv5_block2_3_bn[0][…
conv5_block2_out (Activation)	(None, 1, 1, 2048)	0	conv5_block2_add[0][0]
conv5_block3_1_conv (Conv2D)	(None, 1, 1, 512)	1,049,088	conv5_block2_out[0][0]
conv5_block3_1_bn (BatchNormalization)	(None, 1, 1, 512)	2,048	conv5_block3_1_conv[0…
conv5_block3_1_relu (Activation)	(None, 1, 1, 512)	0	conv5_block3_1_bn[0][…
conv5_block3_2_conv (Conv2D)	(None, 1, 1, 512)	2,359,808	conv5_block3_1_relu[0…
conv5_block3_2_bn (BatchNormalization)	(None, 1, 1, 512)	2,048	conv5_block3_2_conv[0…
conv5_block3_2_relu (Activation)	(None, 1, 1, 512)	0	conv5_block3_2_bn[0][…
conv5_block3_3_conv (Conv2D)	(None, 1, 1, 2048)	1,050,624	conv5_block3_2_relu[0…
conv5_block3_3_bn (BatchNormalization)	(None, 1, 1, 2048)	8,192	conv5_block3_3_conv[0…
conv5_block3_add (Add)	(None, 1, 1, 2048)	0	conv5_block2_out[0][0… conv5_block3_3_bn[0][…
conv5_block3_out (Activation)	(None, 1, 1, 2048)	0	conv5_block3_add[0][0]
global_average_pooling2d… (GlobalAveragePooling2D)	(None, 2048)	0	conv5_block3_out[0][0]
dense_46 (Dense)	(None, 256)	524,544	global_average_poolin…
dense_47 (Dense)	(None, 128)	32,896	dense_46[0][0]
dense_48 (Dense)	(None, 4)	516	dense_47[0][0]

Total params: 24,145,668 (92.11 MB)
Trainable params: 557,956 (2.13 MB)
Non-trainable params: 23,587,712 (89.98 MB)

```
Epoch 1/15
168/168 ━━━━━━━━━━━━━━━━━━━━ 28s 116ms/step - accuracy: 0.5658 - loss: 1.3415 - val_accuracy: 0.7167 - val_loss: 0.7305
Epoch 2/15
168/168 ━━━━━━━━━━━━━━━━━━━━ 34s 91ms/step - accuracy: 0.6993 - loss: 0.7665 - val_accuracy: 0.7158 - val_loss: 0.7135
Epoch 3/15
168/168 ━━━━━━━━━━━━━━━━━━━━ 20s 89ms/step - accuracy: 0.7225 - loss: 0.7097 - val_accuracy: 0.7386 - val_loss: 0.6840
Epoch 4/15
168/168 ━━━━━━━━━━━━━━━━━━━━ 22s 101ms/step - accuracy: 0.7313 - loss: 0.6856 - val_accuracy: 0.7361 - val_loss: 0.6983
Epoch 5/15
168/168 ━━━━━━━━━━━━━━━━━━━━ 19s 89ms/step - accuracy: 0.7414 - loss: 0.6591 - val_accuracy: 0.7511 - val_loss: 0.6563
Epoch 6/15
168/168 ━━━━━━━━━━━━━━━━━━━━ 13s 76ms/step - accuracy: 0.7456 - loss: 0.6487 - val_accuracy: 0.7256 - val_loss: 0.7020
Epoch 7/15
168/168 ━━━━━━━━━━━━━━━━━━━━ 13s 76ms/step - accuracy: 0.7505 - loss: 0.6411 - val_accuracy: 0.7556 - val_loss: 0.6459
Epoch 8/15
168/168 ━━━━━━━━━━━━━━━━━━━━ 13s 76ms/step - accuracy: 0.7587 - loss: 0.6079 - val_accuracy: 0.7572 - val_loss: 0.6435
Epoch 9/15
168/168 ━━━━━━━━━━━━━━━━━━━━ 14s 79ms/step - accuracy: 0.7613 - loss: 0.6142 - val_accuracy: 0.7528 - val_loss: 0.6463
```

```
Epoch 10/15
168/168 ━━━━━━━━━━━━━━━━━━━━ 20s 81ms/step - accuracy: 0.7712 - loss: 0.5843 - val_accuracy: 0.7458 - val_loss: 0.6633
Epoch 11/15
168/168 ━━━━━━━━━━━━━━━━━━━━ 21s 81ms/step - accuracy: 0.7692 - loss: 0.5860 - val_accuracy: 0.7378 - val_loss: 0.7103
Epoch 12/15
168/168 ━━━━━━━━━━━━━━━━━━━━ 16s 89ms/step - accuracy: 0.7696 - loss: 0.5875 - val_accuracy: 0.7522 - val_loss: 0.6585
Epoch 13/15
168/168 ━━━━━━━━━━━━━━━━━━━━ 14s 77ms/step - accuracy: 0.7782 - loss: 0.5660 - val_accuracy: 0.7564 - val_loss: 0.6603
Epoch 14/15
168/168 ━━━━━━━━━━━━━━━━━━━━ 21s 80ms/step - accuracy: 0.7859 - loss: 0.5476 - val_accuracy: 0.7425 - val_loss: 0.7004
Epoch 15/15
168/168 ━━━━━━━━━━━━━━━━━━━━ 16s 91ms/step - accuracy: 0.7862 - loss: 0.5481 - val_accuracy: 0.7458 - val_loss: 0.6728
168/168 ━━━━━━━━━━━━━━━━━━━━ 2s 14ms/step - accuracy: 0.8020 - loss: 0.5024
36/36 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - accuracy: 0.7318 - loss: 0.7021
Performance on the TRAIN set, ACCURACY= 0.8064285516738892
Performance on the TEST set, ACCURACY= 0.7313888669013977
```





[ANSWER] The ResNet50 model achieved a training accuracy of 80.6% and a test accuracy of 73.1% without data augmentation, highlighting its strong feature extraction capabilities due to pre-training on ImageNet. However, the 7.5% accuracy gap between training and test sets indicates some overfitting. The model's performance reflects the power of transfer learning, as the base ResNet50 layers effectively extracted features, requiring minimal fine-tuning of added dense layers. Incorporating data augmentation could further improve generalization and close the performance gap. Despite limited epochs and a frozen base model, ResNet50 efficiently adapted to the CIFAR-4 dataset, showing its robustness for smaller, specialized datasets.

```
#Your code here

#Here we will do some transfer learning with the ResNet50 algorithm, a pre-trained deep convolutional neural network, for our CIFAR-4 dat

from tensorflow.keras.applications import ResNet50
from tensorflow.keras.layers import Dense, Flatten, GlobalAveragePooling2D, Input

datagen = ImageDataGenerator(
    rotation_range=15,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True
)
datagen.fit(X_train)

# (1) DEFINE THE ARCHITECTURE OF MY MODEL
#first, I define all the layers and the way they are connected

# Load the ResNet50 model pre-trained on ImageNet
base_model = ResNet50(weights='imagenet', include_top=False, input_shape=(32, 32, 3))

# Freeze the base model (optional if you only want to train the top layers)
base_model.trainable = False

# Add custom layers for CIFAR-4 classification
inputs = base_model.input
x = base_model.output
x = GlobalAveragePooling2D()(x)  # Pooling to reduce dimensions
x = Dense(256, activation='relu')(x)  # Fully connected layer
x = Dense(128, activation='relu')(x)  # Fully connected layer

outputs = tf.keras.layers.Dense(4, activation='softmax')(x) # my output layer
#Then, I define my model with the input layer, the output layer and a name
my_res50_model = tf.keras.Model(inputs=inputs, outputs=outputs, name="my_res50_model")

#PRINT A SUMMARY OF THE ARCHITECTURE OF MY MODEL WITH THE NUMBER OF TRAINABLE PARAMETERS
my_res50_model.summary()

# (2) DEFINE THE TRAINING HYPER-PARAMETERS WITH THE "COMPILE" METHOD:
'''
(1) Set the "optimizer" [pick 'adam', 'sgd' or 'rmsprop']
(2) Set the loss [cf. lesson #3, we pick the categorical cross-entropy]
(3) Set the final performance metric to evaluate the model
'''

my_res50_model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# (3) NOW, LET'S TRAIN ON MY DATA WITH THE "FIT" METHOD
'''
(1) Set the number of epochs
(2) Set the size of the (mini)batch
(3) Set the training dataset ==> here, X_train with Y_train
(4) Set the validation dataset (X_val, Y_val)
'''
nb_epochs=20
batch_size=100
training_history = my_res50_model.fit(datagen.flow(X_train, Y_train, batch_size=batch_size),
                                      validation_data=(X_val, Y_val),
                                      epochs=nb_epochs)

#COMPUTE THE ACCURACY ON THE TRAINING AND TEST SETS
loss_train, acc_train = my_res50_model.evaluate(X_train, Y_train, batch_size=batch_size)
loss_test, acc_test = my_res50_model.evaluate(X_test, Y_test, batch_size=batch_size)

print("Performance on the TRAIN set, ACCURACY=",acc_train)
print("Performance on the TEST set, ACCURACY=",acc_test)

display_training_curves(training_history)
```

Model: "my_res50_model"

Layer (type)	Output Shape	Param #	Connected to
input_layer_18 (InputLayer)	(None, 32, 32, 3)	0	-
conv1_pad (ZeroPadding2D)	(None, 38, 38, 3)	0	input_layer_18[0][0]
conv1_conv (Conv2D)	(None, 16, 16, 64)	9,472	conv1_pad[0][0]
conv1_bn (BatchNormalization)	(None, 16, 16, 64)	256	conv1_conv[0][0]
conv1_relu (Activation)	(None, 16, 16, 64)	0	conv1_bn[0][0]
pool1_pad (ZeroPadding2D)	(None, 18, 18, 64)	0	conv1_relu[0][0]
pool1_pool (MaxPooling2D)	(None, 8, 8, 64)	0	pool1_pad[0][0]
conv2_block1_1_conv (Conv2D)	(None, 8, 8, 64)	4,160	pool1_pool[0][0]
conv2_block1_1_bn (BatchNormalization)	(None, 8, 8, 64)	256	conv2_block1_1_conv[0…
conv2_block1_1_relu (Activation)	(None, 8, 8, 64)	0	conv2_block1_1_bn[0][…
conv2_block1_2_conv (Conv2D)	(None, 8, 8, 64)	36,928	conv2_block1_1_relu[0…
conv2_block1_2_bn (BatchNormalization)	(None, 8, 8, 64)	256	conv2_block1_2_conv[0…
conv2_block1_2_relu (Activation)	(None, 8, 8, 64)	0	conv2_block1_2_bn[0][…
conv2_block1_0_conv (Conv2D)	(None, 8, 8, 256)	16,640	pool1_pool[0][0]
conv2_block1_3_conv (Conv2D)	(None, 8, 8, 256)	16,640	conv2_block1_2_relu[0…
conv2_block1_0_bn (BatchNormalization)	(None, 8, 8, 256)	1,024	conv2_block1_0_conv[0…
conv2_block1_3_bn (BatchNormalization)	(None, 8, 8, 256)	1,024	conv2_block1_3_conv[0…
conv2_block1_add (Add)	(None, 8, 8, 256)	0	conv2_block1_0_bn[0][… conv2_block1_3_bn[0][…
conv2_block1_out (Activation)	(None, 8, 8, 256)	0	conv2_block1_add[0][0]
conv2_block2_1_conv (Conv2D)	(None, 8, 8, 64)	16,448	conv2_block1_out[0][0]
conv2_block2_1_bn (BatchNormalization)	(None, 8, 8, 64)	256	conv2_block2_1_conv[0…
conv2_block2_1_relu (Activation)	(None, 8, 8, 64)	0	conv2_block2_1_bn[0][…
conv2_block2_2_conv (Conv2D)	(None, 8, 8, 64)	36,928	conv2_block2_1_relu[0…
conv2_block2_2_bn (BatchNormalization)	(None, 8, 8, 64)	256	conv2_block2_2_conv[0…
conv2_block2_2_relu (Activation)	(None, 8, 8, 64)	0	conv2_block2_2_bn[0][…
conv2_block2_3_conv (Conv2D)	(None, 8, 8, 256)	16,640	conv2_block2_2_relu[0…
conv2_block2_3_bn (BatchNormalization)	(None, 8, 8, 256)	1,024	conv2_block2_3_conv[0…
conv2_block2_add (Add)	(None, 8, 8, 256)	0	conv2_block1_out[0][0…