# CNN_Lab

March 10, 2025

## 1 CNN Image Classification Laboration

Images used in this laboration are from CIFAR10. The CIFAR10 dataset contains 60,000 32x32 color images in 10 different classes. The 10 different classes represent airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. There are 6,000 images of each class.

Your task is to make a classifier, using a convolutional neural network, that can correctly classify each image into the correct class.

Complete the code flagged throughout the elaboration and **answer all the questions in the notebook**.

[1]:
```
# Setups
# Automatically reload modules when changed
%reload_ext autoreload
%autoreload 2
```

## 2 Part 1: Convolutions

In the next sections you will familiarize yourself with 2D convolutions.

### 2.1 1.1 What is a convolution?

To understand a bit more about convolutions, we will first test the convolution function in `scipy` using a number of classical filters.

Convolve the image with Gaussian filter, a Sobel X filter, and a Sobel Y filter, using the function `convolve2d` in `signal` from scipy (see the documentation for more details).

In a CNN, many filters are applied in each layer, and the filter coefficients are learned through back propagation (which is in contrast to traditional image processing, where the filters are designed by an expert).

Run the cell below to define a Gaussian filter and a Sobel X and Y filters.

[2]:
```
from scipy import signal
import numpy as np

# Get a test image
from scipy import datasets
```

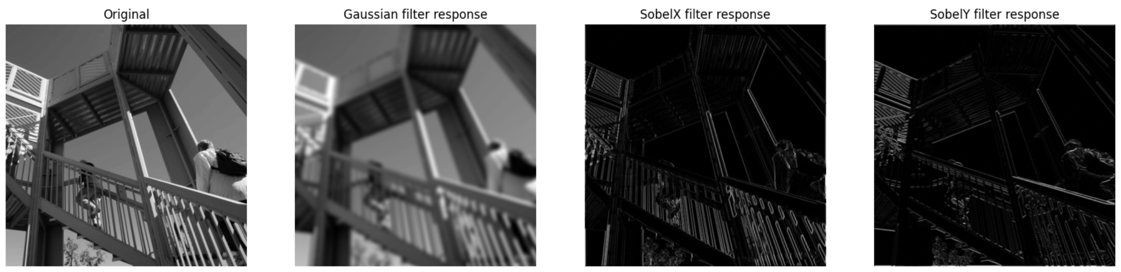
```python
image = datasets.ascent()

# Define a help function for creating a Gaussian filter
def matlab_style_gauss2D(shape=(3,3),sigma=0.5):
    """
    2D gaussian mask - should give the same result as MATLAB's
    fspecial('gaussian',[shape],[sigma])
    """
    m,n = [(ss-1.)/2. for ss in shape]
    y,x = np.ogrid[-m:m+1,-n:n+1]
    h = np.exp( -(x*x + y*y) / (2.*sigma*sigma) )
    h[ h < np.finfo(h.dtype).eps*h.max() ] = 0
    sumh = h.sum()
    if sumh != 0:
        h /= sumh
    return h

# Create Gaussian filter with certain size and standard deviation
gaussFilter = matlab_style_gauss2D((15,15),4)

# Define filter kernels for SobelX and Sobely
sobelX = np.array([[ 1, 0,  -1],
                    [2, 0, -2],
                    [1, 0, -1]])

sobelY = np.array([[ 1, 2,  1],
                    [0, 0, 0],
                    [-1, -2, -1]])
```

[3]:
```python
# Perform convolution using the function 'convolve2d' for the different filters

filterResponseGauss = signal.convolve2d(image, gaussFilter, mode="valid")
filterResponseSobelX = signal.convolve2d(image, sobelX)
filterResponseSobelY = signal.convolve2d(image, sobelY)

# ===========================================
```

[4]:
```python
filterResponseGauss
```

[4]:
```
array([[ 82.87161342,  82.86641452,  82.87018658, …, 117.20774707,
         117.19068118, 117.16521538],
        [ 82.92333321,  82.93096024,  82.95216799, …, 117.24807715,
         117.22729975, 117.19674259],
        [ 83.00159828,  83.02185685,  83.05860547, …, 117.29818266,
         117.27301363, 117.23662516],
        …,
        [174.42461181, 174.40775384, 174.40084864, …,  64.61348445,
         67.94449546,  72.63659477],
       [174.82487385, 174.80609568, 174.79775235, …,  63.65474181,
         65.86887811,  69.44849792],
       [175.24542456, 175.227713  , 175.21933204, …,  61.80283448,
         63.06664188,  65.68779465]])
```

[5]:
```python
import matplotlib.pyplot as plt

# Show filter responses
fig, (ax_orig, ax_filt1, ax_filt2, ax_filt3) = plt.subplots(1, 4, figsize=(20,␣
 ↪6))
ax_orig.imshow(image, cmap='gray')
ax_orig.set_title('Original')
ax_orig.set_axis_off()
ax_filt1.imshow(np.absolute(filterResponseGauss), cmap='gray')
ax_filt1.set_title('Gaussian filter response')
ax_filt1.set_axis_off()
ax_filt2.imshow(np.absolute(filterResponseSobelX), cmap='gray')
ax_filt2.set_title('SobelX filter response')
ax_filt2.set_axis_off()
ax_filt3.imshow(np.absolute(filterResponseSobelY), cmap='gray')
ax_filt3.set_title('SobelY filter response')
ax_filt3.set_axis_off()
```



## 2.2 1.2 Understanding convolutions

**Questions**

1. What do the 3 different filters (Gaussian, SobelX, SobelY) do to the original image?
2. What is the size of the original image? How many channels does it have? How many channels does a color image normally have?
3. What is the size of the different filters?
4. What is the size of the filter response if mode 'same' is used for the convolution ?
5. What is the size of the filter response if mode 'valid' is used for the convolution? How does the size of the valid filter response depend on the size of the filter?

6. Why are ‘valid’ convolutions a problem for CNNs with many layers?

### Answers

1. The gaussian filter applies a gaussian blur, the sobelX filter “detects” vertical edges while sobelY “detects” horizontal edges.
2. The size of the original images is 512 x 512 pixels and it has a singular chanel. A regular color image has three channels (R,G,B).
3. Gaussian: 524 x 524, sabelX and sabelY 514 x 514.
4. The dimensions of the convolution match those of the input image.
5. When using “valid” as an argument, all output pixels which are the result of a convolution that includes zero-padding are removed. Thus, if the input shape is n x n and the filter shape is m x m, the output will be (m - n + 1) x (m - n + 1).
6. When repeating this convolution multiple times, we essentially losed (n - 1) x (n - 1) pixels in every layer, resulting in drastic information loss on the edges of the image.

[6]:
```python
filterResponseGauss = signal.convolve2d(image, gaussFilter)
filterResponseSobelX = signal.convolve2d(image, sobelX)
filterResponseSobelY = signal.convolve2d(image, sobelY)
print("Dimensions with out any mode argument")
print(filterResponseGauss.shape)
print(filterResponseSobelX.shape)
print(filterResponseSobelY.shape)
filterResponseGauss = signal.convolve2d(image, gaussFilter, mode="same")
filterResponseSobelX = signal.convolve2d(image, sobelX, mode="same")
filterResponseSobelY = signal.convolve2d(image, sobelY, mode="same")
print("Dimensions with mode='same'")
print(filterResponseGauss.shape)
print(filterResponseSobelX.shape)
print(filterResponseSobelY.shape)
filterResponseGauss = signal.convolve2d(image, gaussFilter, mode="valid")
filterResponseSobelX = signal.convolve2d(image, sobelX, mode="valid")
filterResponseSobelY = signal.convolve2d(image, sobelY, mode="valid")
print("Dimensions with mode='valid'")
print(filterResponseGauss.shape)
print(filterResponseSobelX.shape)
print(filterResponseSobelY.shape)
```

```
Dimensions with out any mode argument
(526, 526)
(514, 514)
(514, 514)
Dimensions with mode='same'
(512, 512)
(512, 512)
(512, 512)
Dimensions with mode='valid'
(498, 498)
```

```
(510, 510)
(510, 510)
```

## 3 Part 2: Get a graphics card

Skip the next cell if you run on the CPU.

If your computer has a dedicated graphics card and you would like to use it, we need to make sure that our script can see the graphics card that will be used. The graphics cards will perform all the time consuming calculations in every training iteration.

[13]:
```
import os
import warnings

# Ignore FutureWarning from numpy
warnings.simplefilter(action='ignore', category=FutureWarning)

import tensorflow as tf

os.environ["CUDA_DEVICE_ORDER"]="PCI_BUS_ID"

# The GPU id to use, usually either "0" or "1";
os.environ["CUDA_VISIBLE_DEVICES"]="0"

# This sets the GPU to allocate memory only as needed
physical_devices = tf.config.experimental.list_physical_devices('GPU')
if len(physical_devices) != 0:
    tf.config.experimental.set_memory_growth(physical_devices[0], True)
    print("Running on GPU")
else:
    print('No GPU available.')
```

```
Running on GPU
```

### 3.1 How fast is the graphics card?

**Questions**

7. Why are the filters used for a color image of size 7 x 7 x 3, and not 7 x 7 ?
8. What operation is performed by the 'Conv2D' layer? Is it a standard 2D convolution, as performed by the function signal.convolve2d we just tested?
9. Pretend that everyone is using an Nvidia RTX 3090 graphics card, how many CUDA cores does it have? How much memory does the graphics card have?
10. How much memory does the graphics card have?
11. What is stored in the GPU memory while training a CNN?
12. Do you think that a graphics card, compared to the CPU, is equally faster for convolving a batch of 1,000 images, compared to convolving a batch of 3 images? Motivate your answer.

**Answers**

7. Color images are three dimensional, as they have 3 channels. Therefore, the filters must have a third dimension as well.

8. Both perform a convolution on the input but there are still some differences. A convolutional layer allows you to specify a stride value which is how many pixels the filter moves after every application. A high stride value effectively leads to a compression of the image. The filters in the Conv2d-layer are iteratively learned during the training of the model, while the signal.convolve2d applies a static user defined filter. Additionally, signal.convolve2d applies only one filter while the layer can apply multiple filters.

9. According to the NVIDIA homepage the RTX 3090 has 10,496 CUDA cores. It has a memory of 24 GB.

10. It has a memory of 24 GB.

11. The model weights, gradients and the current batch of images for the training are all stored on the GPU.

12. The GPU is not equally faster for every batch size. Loading everything to the graphics card has an overhead that is less relevant for higher batch sizes.

## 4 Part 3: Dataset

In the following section you will load the CIFAR10 dataset, check few samples, perform some preprocessing on the images and the labels, and split the data into training, validation and testing.

### 4.1 3.1 Load the dataset

Run the following section to load the CIFAR10 data, take a total of 10.000 training/validation samples and 2000 testing samples.

[8]:
```
from keras.datasets import cifar10
import numpy as np

classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
 ↪'ship', 'truck']

# Download CIFAR train and test data
(X, Y), (Xtest, Ytest) = cifar10.load_data()

print("Training/validation images have size {} and labels have size {} ".
 ↪format(X.shape, Y.shape))
print("Test images have size {} and labels have size {} \n ".format(Xtest.
 ↪shape, Ytest.shape))

# Reduce the number of images for training/validation and testing to 10000 and␣
 ↪2000 respectively,
# to reduce processing time for this elaboration.
```

```
X = X[0:10000]
Y = Y[0:10000]

Xtest = Xtest[0:2000]
Ytest = Ytest[0:2000]

Ytestint = Ytest

print("Reduced training/validation images have size %s and labels have size %s␣
  ↪" % (X.shape, Y.shape))
print("Reduced test images have size %s and labels have size %s \n" % (Xtest.
  ↪shape, Ytest.shape))

# Check that we have some training examples from each class
for i in range(10):
    print("Number of training/validation examples for class {} is {}" .
  ↪format(i,np.sum(Y == i)))
```

```
Training/validation images have size (50000, 32, 32, 3) and labels have size
(50000, 1)
Test images have size (10000, 32, 32, 3) and labels have size (10000, 1)

Reduced training/validation images have size (10000, 32, 32, 3) and labels have
size (10000, 1)
Reduced test images have size (2000, 32, 32, 3) and labels have size (2000, 1)

Number of training/validation examples for class 0 is 1005
Number of training/validation examples for class 1 is 974
Number of training/validation examples for class 2 is 1032
Number of training/validation examples for class 3 is 1016
Number of training/validation examples for class 4 is 999
Number of training/validation examples for class 5 is 937
Number of training/validation examples for class 6 is 1030
Number of training/validation examples for class 7 is 1001
Number of training/validation examples for class 8 is 1025
Number of training/validation examples for class 9 is 981
```

Lets look at some of the training examples, this cell is already finished. You will see different examples every time you run the cell.

[9]:
```
import matplotlib.pyplot as plt

plt.figure(figsize=(12,4))
for i in range(18):
    idx = np.random.randint(7500)
    label = Y[idx,0]

    plt.subplot(3,6,i+1)
    plt.tight_layout()
    plt.imshow(X[idx])
    plt.title("Class: {} ({})".format(label, classes[label]))
    plt.axis('off')
plt.show()
```

## 4.2 3.2 Split data into training, validation and testing

Split your data (X, Y) into training (Xtrain, Ytrain) and validation (Xval, Yval), so that we have training, validation and test datasets (as in the previous laboration).

We use the `train_test_split` function from scikit learn (see the documentation for more details) to obtain 25% validation set.

[10]:
```
from sklearn.model_selection import train_test_split

# --------------------------------------------
# === Your code here =========================
# --------------------------------------------

# split the original dataset into 75% Training and 25% Validation
Xtrain, Xval, Ytrain, Yval = train_test_split(X, Y, test_size=0.25,␣
 ↪random_state=42)



# Print the size of training data, validation data and test data
print("Training images have shape {} and labels have shape {} ".format(Xtrain.
 ↪shape, Ytrain.shape))
print("Validation images have shape {} and labels have shape {} ".format(Xval.
 ↪shape, Yval.shape))
print("Test images have shape {} and labels have shape {} ".format(Xtest.shape,␣
 ↪Ytest.shape))
# ============================================
Training images have shape (7500, 32, 32, 3) and labels have shape (7500, 1)
Validation images have shape (2500, 32, 32, 3) and labels have shape (2500, 1)
Test images have shape (2000, 32, 32, 3) and labels have shape (2000, 1)
```

## 4.3 3.3 Image Preprocessing

Lets perform some preprocessing. The images are stored as uint8, i.e. 8 bit unsigned integers, but need to be converted to 32 bit floats. We also make sure that the range is -1 to 1, instead of 0 - 255.

[11]:
```python
# Convert datatype for Xtrain, Xval, Xtest, to float32
Xtrain = Xtrain.astype('float32')
Xval = Xval.astype('float32')
Xtest = Xtest.astype('float32')

# Change range of pixel values to [-1,1]
Xtrain = Xtrain / 127.5 - 1
Xval = Xval / 127.5 - 1
Xtest = Xtest / 127.5 - 1
```

## 4.4 3.4 Label preprocessing

The labels (Y) need to be converted from e.g. '4' to "hot encoded", i.e. to a vector of type [0, 0, 0, 1, 0, 0, 0, 0, 0, 0] . We use the `to_categorical`function in Keras (see the documentation for details on how to use it)

[12]:
```python
from tensorflow.keras.utils import to_categorical

# Print shapes before converting the labels
print('Ytrain has size {}.'.format(Ytrain.shape))
print('Yval has size {}.'.format(Yval.shape))
print('Ytest has size {}.'.format(Ytest.shape))


# --------------------------------------------
# === Your code here =========================
# --------------------------------------------

# Your code for converting Ytrain, Yval, Ytest to categorical
Ytrain = to_categorical(Ytrain, 10)
Yval = to_categorical(Yval, 10)
Ytest = to_categorical(Ytest, 10)

# Print shapes after converting the labels
print('Ytrain has shape {}.'.format(Ytrain.shape))
print('Yval has shape {}.'.format(Yval.shape))
print('Ytest has shape {}.'.format(Ytest.shape))
```

```
# ============================================
```

```
Ytrain has size (7500, 1).
Yval has size (2500, 1).
Ytest has size (2000, 1).
Ytrain has shape (7500, 10).
Yval has shape (2500, 10).
Ytest has shape (2000, 10).
```

## 5 Part 4: 2D CNN

In the following sections you will build a 2D CNN model and will train it to perform classification on the CIFAR10 dataset.

### 5.1 4.1 Build CNN model

Start by implementing the `build_CNN` function in the `utilities.py` file. Below you can find the specifications on how your `build_CNN` function should build the model: - Each convolutional layer is composed by: `2D convolution -> batch normalization -> max pooling`. - The `2D convolution` uses a 3 x 3 kernel size, padding='same' and a number of starting filter that is an input to the `build_CNN` function. The number of filters doubles with each convolutional layer (e.g. 32, 64, 128, etc.) - The max pooling layers should have a pool size of 2 x 2. - After the convolutional layers comes a flatten layer, followed by a number of intermediate dense layers. - The number of nodes in the intermediate dense layers before the final dense layer is an input to the `build_CNN` function. The intermediate dense layers use `relu` activation functions and each is followed by `batch normalization`. - The final dense layer should have 10 nodes (=the number of classes in this elaboration) and `softmax` activation.

Here are some relevant functions that you should use in `build_CNN`. For a complete list of functions and their definitions see the keras documentation:

- `model.add()`, adds a layer to the network;
- `Dense()`, a dense network layer. See the documentation what are the input options and outputs of the `Dense()` function.
- `Conv2D()` performs 2D convolutions with a number of filters with a certain size (e.g. 3 x 3) (see documentation).
- `BatchNormalization()`, perform batch normalization (see documentation).
- `MaxPooling2D()`, saves the max for a given pool size, results in down sampling (see documentation).
- `Flatten()`, flatten a multi-channel tensor into a long vector (see documentation).
- `model.compile()`, compiles the model. You can set the input metrics=['accuracy'] to print the classification accuracy during the training.
- cost and loss functions: check the documentation and chose a loss function for binary classification.

To get more information in model compile, training and evaluation see the relevant documentation.

Here you can start with the `Adam` optimizer when compiling the model.

Use the following cell to test your `build_CNN` utility function. Remember to import a relevant cost function for multi-class classification from keras.losses which relates to how many classes you have.

[15]:
```
Xtrain.shape
```

[15]:
```
(7500, 32, 32, 3)
```

[13]:
```
# import utilities
from utilities import build_CNN

# ---------------------------------------------
# === Your code here =========================
# ---------------------------------------------

# import a suitable loss function from keras.losses and use as input to the␣
  ↪build_DNN function.
from tf_keras.losses import CategoricalCrossentropy

# Build a DNN model following the specifications above
model = build_CNN(input_shape=Xtrain.shape[1:],loss=CategoricalCrossentropy)

# =============================================
```

```
I0000 00:00:1741337642.753228     796 gpu_device.cc:2022] Created device
/job:localhost/replica:0/task:0/device:GPU:0 with 5563 MB memory:  -> device: 0,
name: NVIDIA GeForce RTX 4060 Laptop GPU, pci bus id: 0000:01:00.0, compute
capability: 8.9
```

## 5.2 4.2 Train 2D CNN

Time to train the CNN!

Start with a model with 2 convolutional layers where the first layer has have 16 filters, and with no intermediate dense layers.

Set the training parameters, build the model and run the training.

Use the following training parameters: - `batch_size=20` - `epochs=20` - `learning_rate=0.01`

Relevant functions: - `build_CNN`, the function that you defined in the `utilities.py` file. - `model.fit()`, train the model with some training data (see documentation). - `model.evaluate()`, apply the trained model to some test data (see documentation).

## 5.3 2 convolutional layers, no intermediate dense layers

[19]:
```
# ---------------------------------------------
# === Your code here =========================
# ---------------------------------------------


# Setup some training parameters
```

```
batch_size = 20
epochs = 20
input_shape = Xtrain.shape[1:]
learning_rate = 0.01

# Build model
model1 = build_CNN(input_shape=input_shape, loss="categorical_crossentropy",␣
↪n_conv_layers=2, n_dense_layers=0, n_filters=16)

# Train the model  using training data and validation data
history1 = model1.fit(Xtrain, Ytrain, batch_size=batch_size, epochs=epochs,␣
↪validation_data=(Xval, Yval))

# ============================================
```

```
Epoch 1/20
375/375 [==============================] - 5s 10ms/step - loss: 1.9146 -
accuracy: 0.3917 - val_loss: 1.7899 - val_accuracy: 0.3676
Epoch 2/20
375/375 [==============================] - 3s 8ms/step - loss: 1.3563 -
accuracy: 0.5224 - val_loss: 1.4288 - val_accuracy: 0.5040
Epoch 3/20
375/375 [==============================] - 3s 8ms/step - loss: 1.2127 -
accuracy: 0.5745 - val_loss: 1.3232 - val_accuracy: 0.5296
Epoch 4/20
375/375 [==============================] - 3s 9ms/step - loss: 1.1091 -
accuracy: 0.6047 - val_loss: 1.3414 - val_accuracy: 0.5320
Epoch 5/20
375/375 [==============================] - 3s 9ms/step - loss: 0.9942 -
accuracy: 0.6456 - val_loss: 1.3573 - val_accuracy: 0.5408
Epoch 6/20
375/375 [==============================] - 3s 8ms/step - loss: 0.9207 -
accuracy: 0.6755 - val_loss: 1.3478 - val_accuracy: 0.5708
Epoch 7/20
375/375 [==============================] - 3s 8ms/step - loss: 0.8497 -
accuracy: 0.6987 - val_loss: 1.4025 - val_accuracy: 0.5464
Epoch 8/20
375/375 [==============================] - 3s 8ms/step - loss: 0.7828 -
accuracy: 0.7213 - val_loss: 1.4760 - val_accuracy: 0.5376
Epoch 9/20
375/375 [==============================] - 3s 8ms/step - loss: 0.7167 -
accuracy: 0.7509 - val_loss: 1.6421 - val_accuracy: 0.5244
Epoch 10/20
375/375 [==============================] - 3s 8ms/step - loss: 0.6509 -
accuracy: 0.7664 - val_loss: 1.5538 - val_accuracy: 0.5492
Epoch 11/20
375/375 [==============================] - 2s 6ms/step - loss: 0.5976 -
```

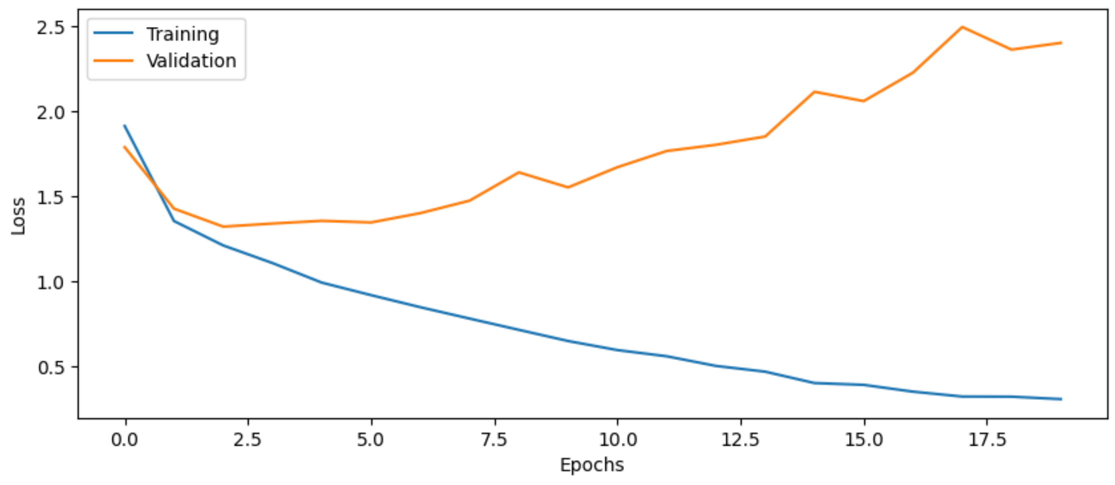
```
accuracy: 0.7931 - val_loss: 1.6725 - val_accuracy: 0.5356
Epoch 12/20
375/375 [==============================] - 3s 8ms/step - loss: 0.5613 -
accuracy: 0.8033 - val_loss: 1.7677 - val_accuracy: 0.5132
Epoch 13/20
375/375 [==============================] - 3s 8ms/step - loss: 0.5046 -
accuracy: 0.8181 - val_loss: 1.8042 - val_accuracy: 0.5392
Epoch 14/20
375/375 [==============================] - 3s 8ms/step - loss: 0.4708 -
accuracy: 0.8297 - val_loss: 1.8528 - val_accuracy: 0.5376
Epoch 15/20
375/375 [==============================] - 3s 8ms/step - loss: 0.4039 -
accuracy: 0.8531 - val_loss: 2.1157 - val_accuracy: 0.5208
Epoch 16/20
375/375 [==============================] - 3s 8ms/step - loss: 0.3932 -
accuracy: 0.8609 - val_loss: 2.0610 - val_accuracy: 0.5444
Epoch 17/20
375/375 [==============================] - 3s 8ms/step - loss: 0.3533 -
accuracy: 0.8736 - val_loss: 2.2283 - val_accuracy: 0.5252
Epoch 18/20
375/375 [==============================] - 3s 8ms/step - loss: 0.3246 -
accuracy: 0.8851 - val_loss: 2.4965 - val_accuracy: 0.5164
Epoch 19/20
375/375 [==============================] - 3s 8ms/step - loss: 0.3237 -
accuracy: 0.8837 - val_loss: 2.3637 - val_accuracy: 0.5412
Epoch 20/20
375/375 [==============================] - 3s 8ms/step - loss: 0.3096 -
accuracy: 0.8873 - val_loss: 2.4033 - val_accuracy: 0.5448
```

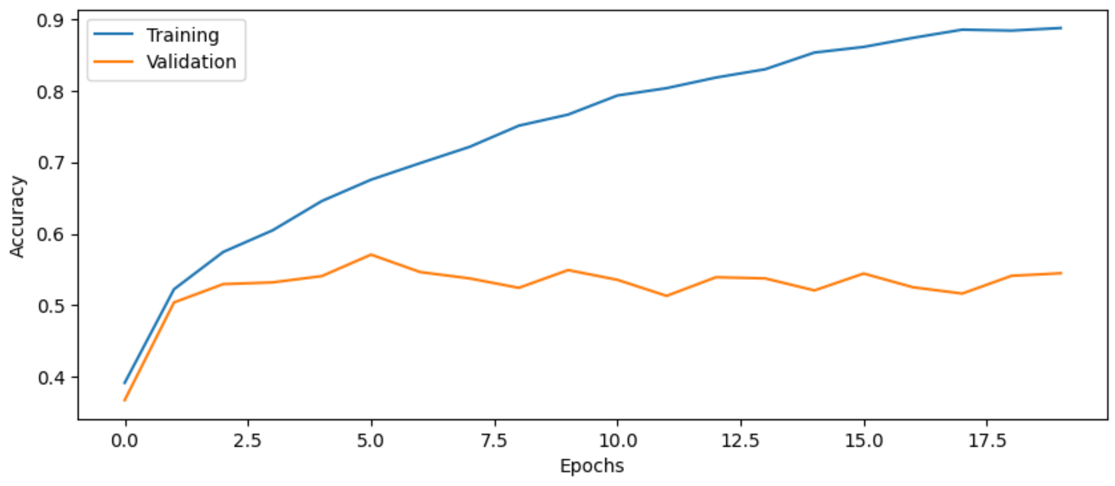
[20]:
```
# --------------------------------------------
# === Your code here =========================
# --------------------------------------------

# Evaluate the trained model on test set, not used in training or validation
score = model1.evaluate(Xtest, Ytest, batch_size=batch_size)

# ============================================

print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])
```

```
100/100 [==============================] - 0s 5ms/step - loss: 2.5488 -
accuracy: 0.5230
Test loss: 2.5488
Test accuracy: 0.5230
```

[21]:
```
from utilities import plot_results
# Plot the history from the training run
```

```
plot_results(history1)
```





[22]:
```
score_test=model1.evaluate(Xtrain, Ytrain, batch_size=batch_size)
print('Train loss: %.4f' % score_test[0])
print('Train accuracy: %.4f' % score_test[1])
```

```
375/375 [==============================] - 1s 3ms/step - loss: 0.1910 -
accuracy: 0.9324
Train loss: 0.1910
Train accuracy: 0.9324
```

```
CarbonTracker: WARNING - ElectricityMaps API key not set. Will default to
average carbon intensity.
CarbonTracker: WARNING - Failed to retrieve carbon intensity: Defaulting to
```

```
average carbon intensity 40.694878 gCO2/kWh.
CarbonTracker: WARNING - ElectricityMaps API key not set. Will default to
average carbon intensity.
CarbonTracker: WARNING - Failed to retrieve carbon intensity: Defaulting to
average carbon intensity 40.694878 gCO2/kWh.
CarbonTracker: WARNING - ElectricityMaps API key not set. Will default to
average carbon intensity.
CarbonTracker: WARNING - Failed to retrieve carbon intensity: Defaulting to
average carbon intensity 40.694878 gCO2/kWh.
CarbonTracker: WARNING - ElectricityMaps API key not set. Will default to
average carbon intensity.
CarbonTracker: WARNING - Failed to retrieve carbon intensity: Defaulting to
average carbon intensity 40.694878 gCO2/kWh.
CarbonTracker: WARNING - ElectricityMaps API key not set. Will default to
average carbon intensity.
CarbonTracker: WARNING - Failed to retrieve carbon intensity: Defaulting to
average carbon intensity 40.694878 gCO2/kWh.
CarbonTracker: WARNING - ElectricityMaps API key not set. Will default to
average carbon intensity.
CarbonTracker: WARNING - Failed to retrieve carbon intensity: Defaulting to
average carbon intensity 40.694878 gCO2/kWh.
CarbonTracker: WARNING - ElectricityMaps API key not set. Will default to
average carbon intensity.
CarbonTracker: WARNING - Failed to retrieve carbon intensity: Defaulting to
average carbon intensity 40.694878 gCO2/kWh.
CarbonTracker: WARNING - ElectricityMaps API key not set. Will default to
average carbon intensity.
CarbonTracker: WARNING - Failed to retrieve carbon intensity: Defaulting to
average carbon intensity 40.694878 gCO2/kWh.
```

## 5.4 4.3 Improving model performance

Write down the test accuracy, are you satisfied with the classifier performance (random chance is 10%)?

#### Questions

13. How big is the difference between training and test accuracy?
14. For the DNN elaboration we used a batch size of 10.000, why do we need to use a smaller batch size in this elaboration?

#### Answers

13. The difference is quiet big. The train accuracy is 0.8955 and the test accuracy is 0.5660. That makes a difference of 0.3295. The big difference is an indicator for overfitting.
14. Images require more space in memory, so we cannot load as many datapoints at the same time. Additionally, the dataset we use this time is smaller. Using a big batch size would lead to fewer weight updates which could be unfavourable.

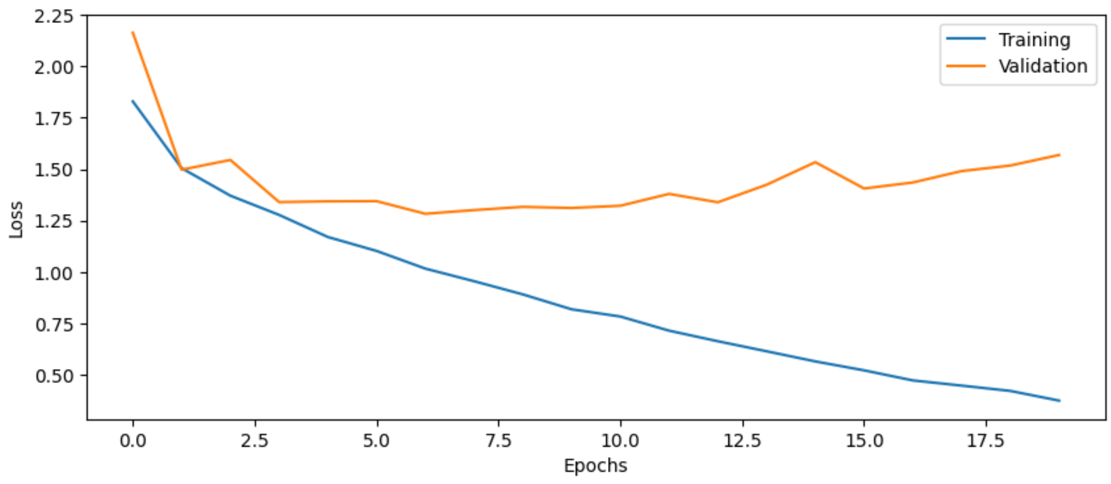
Experiment with several model configurations in the following sections.

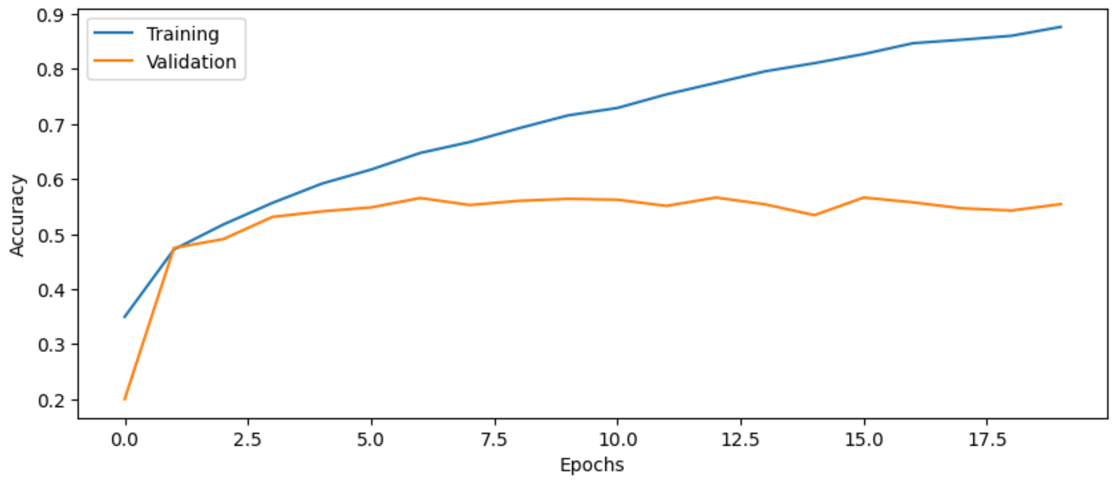
### 5.4.1 2 convolutional layers with 16 starting filters and 1 intermediate dense layer (50 nodes)

```
[20]: # --------------------------------------------
# === Your code here =========================
# --------------------------------------------
n_conv_layers = 2
n_dense_layers = 1
n_filters = 16
n_nodes = 50

# Build and train model
model1 = build_CNN(input_shape=input_shape, loss="categorical_crossentropy",␣
 ↪n_conv_layers=n_conv_layers, n_dense_layers=n_dense_layers,␣
 ↪n_filters=n_filters)

history1 = model1.fit(Xtrain, Ytrain, batch_size=batch_size, epochs=epochs,␣
 ↪validation_data=(Xval, Yval))

# Evaluate model on test data
score = model1.evaluate(Xtest, Ytest, batch_size=batch_size)

# ============================================

print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])

# Plot the history from the training run
plot_results(history1)
```

```
Epoch 1/20
375/375 [==============================] - 5s 10ms/step - loss: 1.8293 -
accuracy: 0.3496 - val_loss: 2.1635 - val_accuracy: 0.2000
Epoch 2/20
375/375 [==============================] - 3s 9ms/step - loss: 1.5048 -
accuracy: 0.4720 - val_loss: 1.4972 - val_accuracy: 0.4748
Epoch 3/20
375/375 [==============================] - 3s 9ms/step - loss: 1.3710 -
accuracy: 0.5173 - val_loss: 1.5448 - val_accuracy: 0.4908
Epoch 4/20
375/375 [==============================] - 3s 9ms/step - loss: 1.2774 -
accuracy: 0.5567 - val_loss: 1.3397 - val_accuracy: 0.5312
Epoch 5/20
375/375 [==============================] - 3s 8ms/step - loss: 1.1699 -
accuracy: 0.5916 - val_loss: 1.3434 - val_accuracy: 0.5412
Epoch 6/20
```

```
375/375 [==============================] - 3s 9ms/step - loss: 1.1021 -
accuracy: 0.6172 - val_loss: 1.3443 - val_accuracy: 0.5484
Epoch 7/20
375/375 [==============================] - 3s 8ms/step - loss: 1.0162 -
accuracy: 0.6475 - val_loss: 1.2830 - val_accuracy: 0.5656
Epoch 8/20
375/375 [==============================] - 3s 8ms/step - loss: 0.9549 -
accuracy: 0.6672 - val_loss: 1.3009 - val_accuracy: 0.5528
Epoch 9/20
375/375 [==============================] - 3s 8ms/step - loss: 0.8908 -
accuracy: 0.6923 - val_loss: 1.3166 - val_accuracy: 0.5604
Epoch 10/20
375/375 [==============================] - 3s 8ms/step - loss: 0.8183 -
accuracy: 0.7157 - val_loss: 1.3111 - val_accuracy: 0.5640
Epoch 11/20
375/375 [==============================] - 3s 8ms/step - loss: 0.7832 -
accuracy: 0.7292 - val_loss: 1.3220 - val_accuracy: 0.5624
Epoch 12/20
375/375 [==============================] - 3s 8ms/step - loss: 0.7144 -
accuracy: 0.7540 - val_loss: 1.3797 - val_accuracy: 0.5512
Epoch 13/20
375/375 [==============================] - 3s 8ms/step - loss: 0.6629 -
accuracy: 0.7748 - val_loss: 1.3388 - val_accuracy: 0.5664
Epoch 14/20
375/375 [==============================] - 3s 8ms/step - loss: 0.6141 -
accuracy: 0.7959 - val_loss: 1.4241 - val_accuracy: 0.5540
Epoch 15/20
375/375 [==============================] - 3s 8ms/step - loss: 0.5649 -
accuracy: 0.8107 - val_loss: 1.5339 - val_accuracy: 0.5344
Epoch 16/20
375/375 [==============================] - 3s 8ms/step - loss: 0.5214 -
accuracy: 0.8272 - val_loss: 1.4059 - val_accuracy: 0.5664
Epoch 17/20
375/375 [==============================] - 3s 8ms/step - loss: 0.4724 -
accuracy: 0.8469 - val_loss: 1.4351 - val_accuracy: 0.5576
Epoch 18/20
375/375 [==============================] - 3s 8ms/step - loss: 0.4472 -
accuracy: 0.8533 - val_loss: 1.4905 - val_accuracy: 0.5468
Epoch 19/20
375/375 [==============================] - 3s 8ms/step - loss: 0.4213 -
accuracy: 0.8604 - val_loss: 1.5174 - val_accuracy: 0.5428
Epoch 20/20
375/375 [==============================] - 3s 8ms/step - loss: 0.3742 -
accuracy: 0.8767 - val_loss: 1.5689 - val_accuracy: 0.5544
100/100 [==============================] - 1s 5ms/step - loss: 1.5267 -
accuracy: 0.5540
Test loss: 1.5267
Test accuracy: 0.5540
```





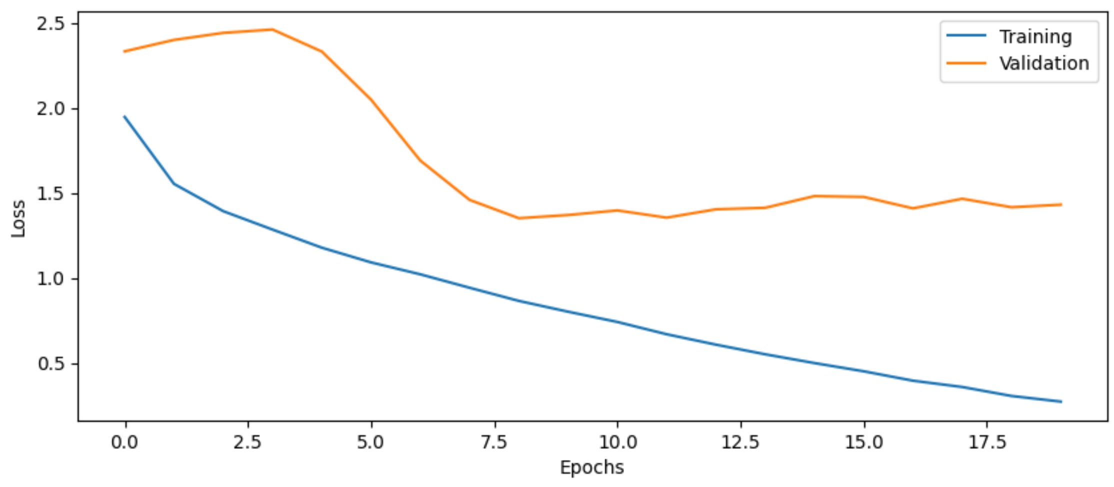
### 5.4.2 4 convolutional layers with 16 starting filters and 1 intermediate dense layer (50 nodes)

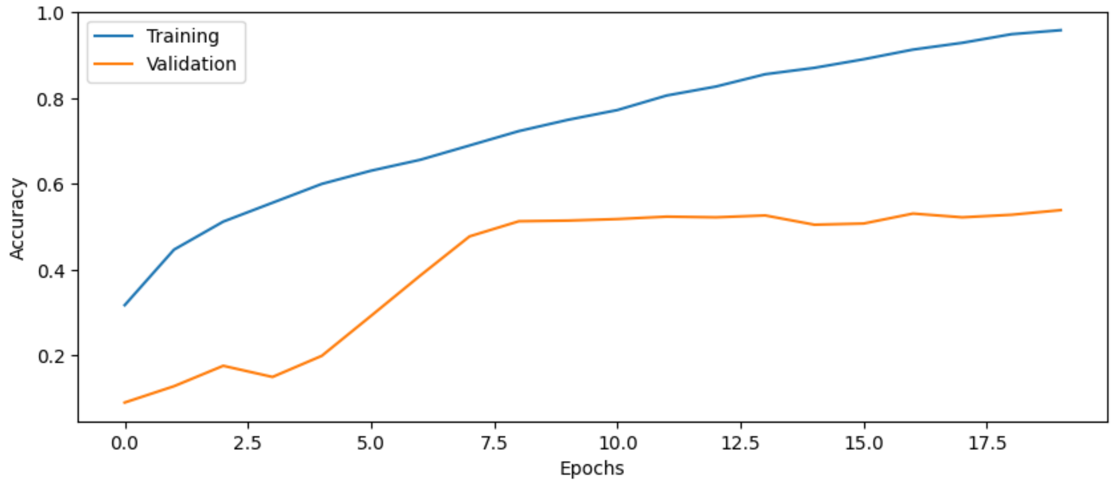
[28]:
```
# --------------------------------------------
# === Your code here =========================
# --------------------------------------------
batch_size = 100
n_conv_layers = 4
n_dense_layers = 1
n_filters = 16
n_nodes = 50

# Build and train model
```

```
model2 = build_CNN(input_shape=input_shape, loss="categorical_crossentropy",␣
 ↪n_conv_layers=n_conv_layers, n_dense_layers=n_dense_layers,␣
 ↪n_filters=n_filters)

history2 = model2.fit(Xtrain, Ytrain, batch_size=batch_size, epochs=epochs,␣
 ↪validation_data=(Xval, Yval))

# Evaluate model on test data
score = model2.evaluate(Xtest, Ytest, batch_size=batch_size)

# ============================================

print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])

# Plot the history from the training run
plot_results(history2)
```

```
Epoch 1/20
75/75 [==============================] - 3s 17ms/step - loss: 1.9457 - accuracy:
0.3177 - val_loss: 2.3313 - val_accuracy: 0.0908
Epoch 2/20
75/75 [==============================] - 1s 10ms/step - loss: 1.5527 - accuracy:
0.4471 - val_loss: 2.3984 - val_accuracy: 0.1288
Epoch 3/20
75/75 [==============================] - 1s 11ms/step - loss: 1.3923 - accuracy:
0.5124 - val_loss: 2.4398 - val_accuracy: 0.1764
Epoch 4/20
75/75 [==============================] - 1s 11ms/step - loss: 1.2842 - accuracy:
0.5564 - val_loss: 2.4595 - val_accuracy: 0.1504
Epoch 5/20
75/75 [==============================] - 1s 10ms/step - loss: 1.1779 - accuracy:
0.6003 - val_loss: 2.3301 - val_accuracy: 0.1996
Epoch 6/20
75/75 [==============================] - 1s 11ms/step - loss: 1.0913 - accuracy:
0.6311 - val_loss: 2.0476 - val_accuracy: 0.2928
Epoch 7/20
75/75 [==============================] - 1s 11ms/step - loss: 1.0213 - accuracy:
0.6565 - val_loss: 1.6895 - val_accuracy: 0.3868
Epoch 8/20
75/75 [==============================] - 1s 11ms/step - loss: 0.9424 - accuracy:
0.6899 - val_loss: 1.4585 - val_accuracy: 0.4780
Epoch 9/20
75/75 [==============================] - 1s 12ms/step - loss: 0.8652 - accuracy:
0.7233 - val_loss: 1.3510 - val_accuracy: 0.5132
Epoch 10/20
75/75 [==============================] - 1s 12ms/step - loss: 0.8017 - accuracy:
```

```
0.7497 - val_loss: 1.3697 - val_accuracy: 0.5148
Epoch 11/20
75/75 [==============================] - 1s 12ms/step - loss: 0.7414 - accuracy:
0.7724 - val_loss: 1.3966 - val_accuracy: 0.5184
Epoch 12/20
75/75 [==============================] - 1s 12ms/step - loss: 0.6692 - accuracy:
0.8063 - val_loss: 1.3542 - val_accuracy: 0.5240
Epoch 13/20
75/75 [==============================] - 1s 11ms/step - loss: 0.6082 - accuracy:
0.8272 - val_loss: 1.4033 - val_accuracy: 0.5224
Epoch 14/20
75/75 [==============================] - 1s 11ms/step - loss: 0.5515 - accuracy:
0.8559 - val_loss: 1.4121 - val_accuracy: 0.5268
Epoch 15/20
75/75 [==============================] - 1s 11ms/step - loss: 0.4999 - accuracy:
0.8708 - val_loss: 1.4809 - val_accuracy: 0.5052
Epoch 16/20
75/75 [==============================] - 1s 11ms/step - loss: 0.4516 - accuracy:
0.8908 - val_loss: 1.4760 - val_accuracy: 0.5080
Epoch 17/20
75/75 [==============================] - 1s 12ms/step - loss: 0.3963 - accuracy:
0.9133 - val_loss: 1.4088 - val_accuracy: 0.5312
Epoch 18/20
75/75 [==============================] - 1s 11ms/step - loss: 0.3594 - accuracy:
0.9291 - val_loss: 1.4652 - val_accuracy: 0.5224
Epoch 19/20
75/75 [==============================] - 1s 11ms/step - loss: 0.3066 - accuracy:
0.9492 - val_loss: 1.4155 - val_accuracy: 0.5284
Epoch 20/20
75/75 [==============================] - 1s 12ms/step - loss: 0.2736 - accuracy:
0.9585 - val_loss: 1.4304 - val_accuracy: 0.5392
20/20 [==============================] - 0s 6ms/step - loss: 1.4433 - accuracy:
0.5240
Test loss: 1.4433
Test accuracy: 0.5240
```





[30]:
```
model2.summary()
```

```
Model: "sequential_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_10 (Conv2D)          (None, 32, 32, 16)        448

 batch_normalization_11 (Ba  (None, 32, 32, 16)        64
 tchNormalization)

 max_pooling2d_10 (MaxPooli  (None, 16, 16, 16)        0
 ng2D)
 conv2d_11 (Conv2D)          (None, 16, 16, 32)        4640

 batch_normalization_12 (Ba  (None, 16, 16, 32)        128
 tchNormalization)

 max_pooling2d_11 (MaxPooli  (None, 8, 8, 32)          0
 ng2D)

 conv2d_12 (Conv2D)          (None, 8, 8, 64)          18496

 batch_normalization_13 (Ba  (None, 8, 8, 64)          256
 tchNormalization)

 max_pooling2d_12 (MaxPooli  (None, 4, 4, 64)          0
 ng2D)

 conv2d_13 (Conv2D)          (None, 4, 4, 128)         73856

 batch_normalization_14 (Ba  (None, 4, 4, 128)         512
 tchNormalization)

 max_pooling2d_13 (MaxPooli  (None, 2, 2, 128)         0
 ng2D)

 flatten_5 (Flatten)         (None, 512)               0

 dense_6 (Dense)             (None, 50)                25650

 batch_normalization_15 (Ba  (None, 50)                200
 tchNormalization)

 dense_7 (Dense)             (None, 10)                510

=================================================================
Total params: 124760 (487.34 KB)
Trainable params: 124180 (485.08 KB)
Non-trainable params: 580 (2.27 KB)
_________________________________________________________________
```

[33]:
```
9*32*16+32
```

[33]:
```
4640
```

## 5.5 4.4 Plot the CNN architecture and understand the internal model dimensions

To understand your network better, print the architecture using `model.summary()`

**Questions**

15. How many trainable parameters does your network have? Which part of the network contains most of the parameters?
16. What is the input to and output of a Conv2D layer? What are the dimensions of the input and output?
17. Is the batch size always the first dimension of each 4D tensor? Check the documentation for Conv2D.
18. If a convolutional layer that contains 128 filters is applied to an input with 32 channels, what is the number of channels in the output?
19. Why is the number of parameters in each Conv2D layer *not* equal to the number of filters times the number of filter coefficients per filter (plus biases)?
20. How does MaxPooling help in reducing the number of parameters to train?

**Answers**

15. The model has 124,180 trainable parameters in total. The last convolutional layer with 128 filters has the most trainable parameters with 73,856.
16. If you define the output size as (x,y,z), the output has the same x and y as the input and the number of filters as z. This applies to the case of using “same”-padding.
17. According to the documentation there are two ways to define the input tensors but the batch size is always the first dimension. They only differ in the position of the channel, which can be before or after hight and width.
18. The number of channels in the output is always the number of filters in the layer, therefore the number of channels in the output is 128 in this case.
19. We have to consider the number of channels as well. The number of parameters is `number_of_filter_coefficients * number_of_filters * number_of_channels + bias`
20. MaxPooling reduces the hight and width depending on the used kernel size and therefore reduces also the number of trainable parameters.

## 5.6 4.5 Dropout regularization

Add dropout regularization between each intermediate dense layer, with dropout probability 50%.

**Questions**

21. How much did the test accuracy improve with dropout, compared to without dropout?
22. What other types of regularization can be applied? How can you add `L2 regularization` for the convolutional layers?

**Answers**

21. Without changing any other hyperparameters, the model is still heavily overfitting but we can improve the test accuracy to 0.5615.
22. Keras offers the option to use L1 and L2 regularization but also custom regularizers inheriting from its Regularizer class. L2 regularization can be added as parameter in the Conv2D layers. It can be used to add a penalty on the layer's kernel, the layer's bias or the layer's output.

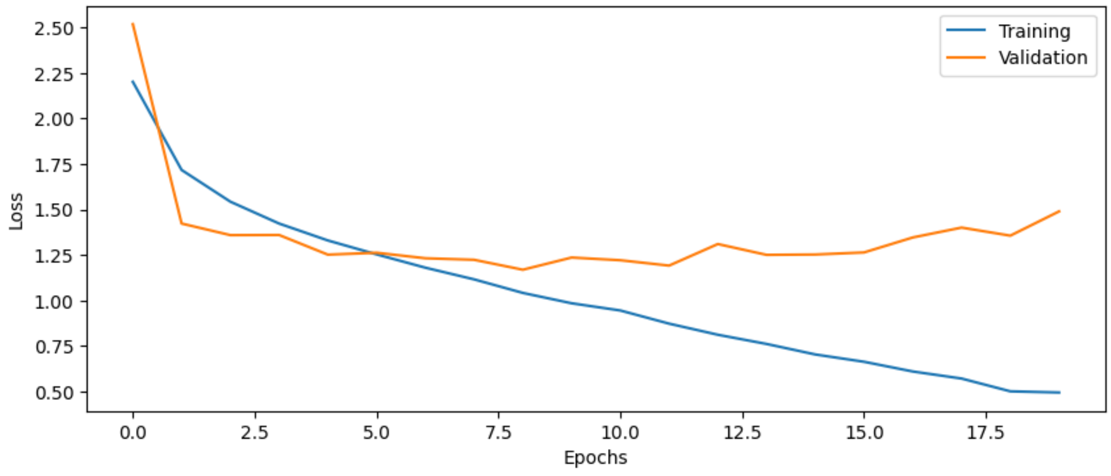
### 5.6.1 4 convolutional layers with 16 starting filters and 1 intermediate dense layer (50 nodes) with dropout

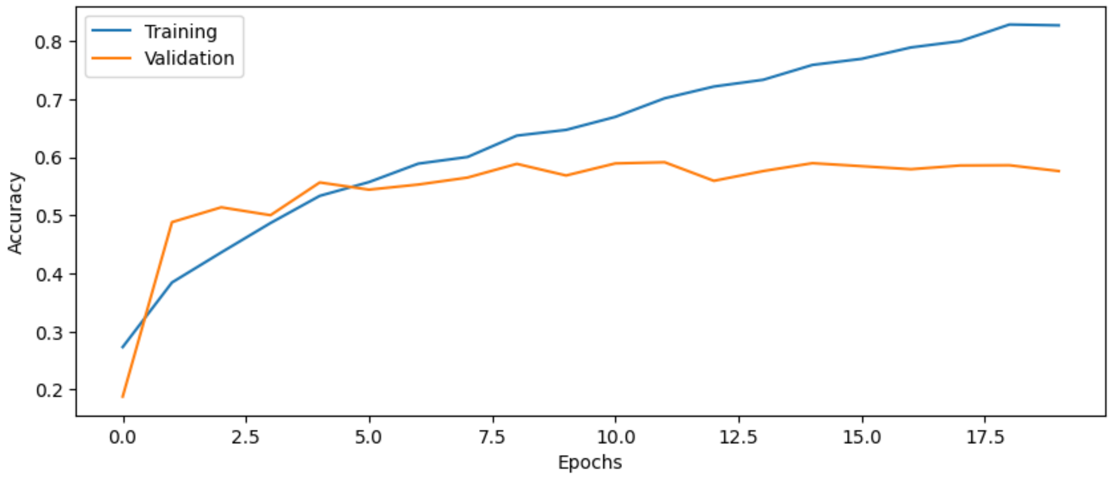
[39]:
```
# -------------------------------------------
# === Your code here ========================
# -------------------------------------------

# Setup some training parameters
batch_size = 20
epochs = 20
input_shape = Xtrain.shape[1:]
learning_rate = 0.01

n_conv_layers = 4
n_dense_layers = 1
n_filters = 16
n_nodes = 50
use_dropout = True

# Build and train model
model3 = build_CNN(input_shape=input_shape, loss="categorical_crossentropy",␣
 ↪n_conv_layers=n_conv_layers, n_dense_layers=n_dense_layers,␣
 ↪n_filters=n_filters, use_dropout=use_dropout)

history3 = model3.fit(Xtrain, Ytrain, batch_size=batch_size, epochs=epochs,␣
 ↪validation_data=(Xval, Yval))

# Evaluate model on test data
score = model3.evaluate(Xtest, Ytest, batch_size=batch_size)

# ===========================================

print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])

# Plot the history from the training run
plot_results(history3)
```

```
Epoch 1/20
375/375 [==============================] - 3s 7ms/step - loss: 2.2012 -
```

```
accuracy: 0.2733 - val_loss: 2.5177 - val_accuracy: 0.1880
Epoch 2/20
375/375 [==============================] - 2s 6ms/step - loss: 1.7173 -
accuracy: 0.3845 - val_loss: 1.4230 - val_accuracy: 0.4884
Epoch 3/20
375/375 [==============================] - 2s 6ms/step - loss: 1.5431 -
accuracy: 0.4363 - val_loss: 1.3594 - val_accuracy: 0.5140
Epoch 4/20
375/375 [==============================] - 2s 6ms/step - loss: 1.4230 -
accuracy: 0.4869 - val_loss: 1.3601 - val_accuracy: 0.5004
Epoch 5/20
375/375 [==============================] - 2s 6ms/step - loss: 1.3301 -
accuracy: 0.5337 - val_loss: 1.2522 - val_accuracy: 0.5568
Epoch 6/20
375/375 [==============================] - 2s 6ms/step - loss: 1.2537 -
accuracy: 0.5575 - val_loss: 1.2626 - val_accuracy: 0.5444
Epoch 7/20
375/375 [==============================] - 2s 6ms/step - loss: 1.1807 -
accuracy: 0.5892 - val_loss: 1.2323 - val_accuracy: 0.5532
Epoch 8/20
375/375 [==============================] - 2s 6ms/step - loss: 1.1165 -
accuracy: 0.6007 - val_loss: 1.2240 - val_accuracy: 0.5652
Epoch 9/20
375/375 [==============================] - 2s 6ms/step - loss: 1.0421 -
accuracy: 0.6375 - val_loss: 1.1694 - val_accuracy: 0.5888
Epoch 10/20
375/375 [==============================] - 2s 6ms/step - loss: 0.9852 -
accuracy: 0.6473 - val_loss: 1.2363 - val_accuracy: 0.5688
Epoch 11/20
375/375 [==============================] - 2s 7ms/step - loss: 0.9456 -
accuracy: 0.6697 - val_loss: 1.2215 - val_accuracy: 0.5896
Epoch 12/20
375/375 [==============================] - 2s 6ms/step - loss: 0.8733 -
accuracy: 0.7017 - val_loss: 1.1921 - val_accuracy: 0.5916
Epoch 13/20
375/375 [==============================] - 2s 7ms/step - loss: 0.8124 -
accuracy: 0.7220 - val_loss: 1.3105 - val_accuracy: 0.5596
Epoch 14/20
375/375 [==============================] - 2s 7ms/step - loss: 0.7618 -
accuracy: 0.7336 - val_loss: 1.2510 - val_accuracy: 0.5764
Epoch 15/20
375/375 [==============================] - 2s 6ms/step - loss: 0.7041 -
accuracy: 0.7592 - val_loss: 1.2530 - val_accuracy: 0.5900
Epoch 16/20
375/375 [==============================] - 2s 7ms/step - loss: 0.6639 -
accuracy: 0.7697 - val_loss: 1.2643 - val_accuracy: 0.5848
Epoch 17/20
375/375 [==============================] - 2s 7ms/step - loss: 0.6106 -
```

```
accuracy: 0.7893 - val_loss: 1.3466 - val_accuracy: 0.5796
Epoch 18/20
375/375 [==============================] - 2s 6ms/step - loss: 0.5718 -
accuracy: 0.8001 - val_loss: 1.4007 - val_accuracy: 0.5860
Epoch 19/20
375/375 [==============================] - 2s 7ms/step - loss: 0.5017 -
accuracy: 0.8288 - val_loss: 1.3565 - val_accuracy: 0.5864
Epoch 20/20
375/375 [==============================] - 2s 6ms/step - loss: 0.4952 -
accuracy: 0.8273 - val_loss: 1.4897 - val_accuracy: 0.5764
100/100 [==============================] - 0s 3ms/step - loss: 1.5405 -
accuracy: 0.5615
Test loss: 1.5405
Test accuracy: 0.5615
```





## 5.7 4.6 Tweaking model performance

You have now seen the basic building blocks of a 2D CNN. To further improve performance involves changing the number of convolutional layers, the number of filters per layer, the number of intermediate dense layers, the number of nodes in the intermediate dense layers, batch size, learning rate, number of epochs, etc. Spend some time (30 - 90 minutes) testing different settings.

### Questions

23. How high test accuracy can you obtain? What is your best configuration?

### Answers

23. Within the time span a test accuracy of 0.5795 is the best that we can achieve. We obeserved that the performence of bigger networks gets worse and tends to overfit more.

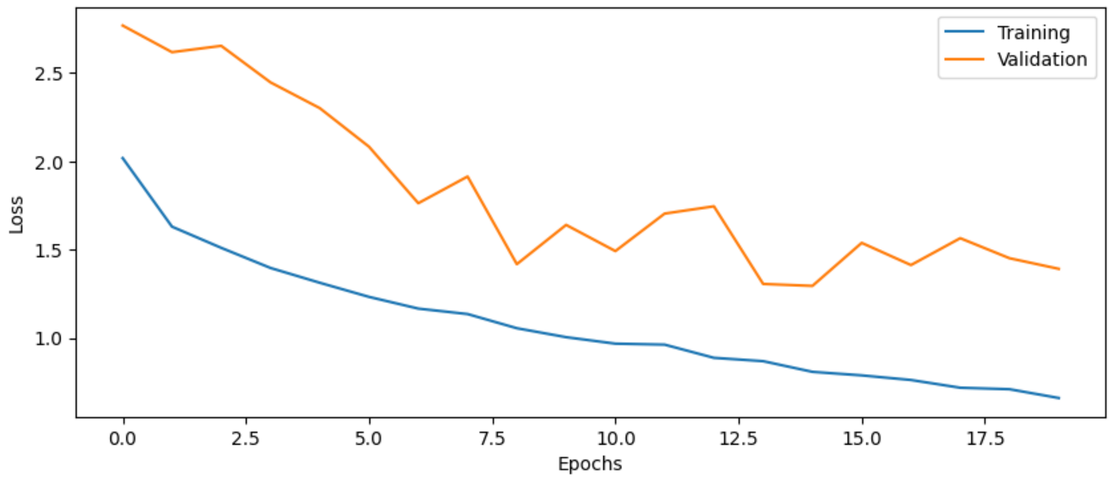
## 5.8 *Your best config*

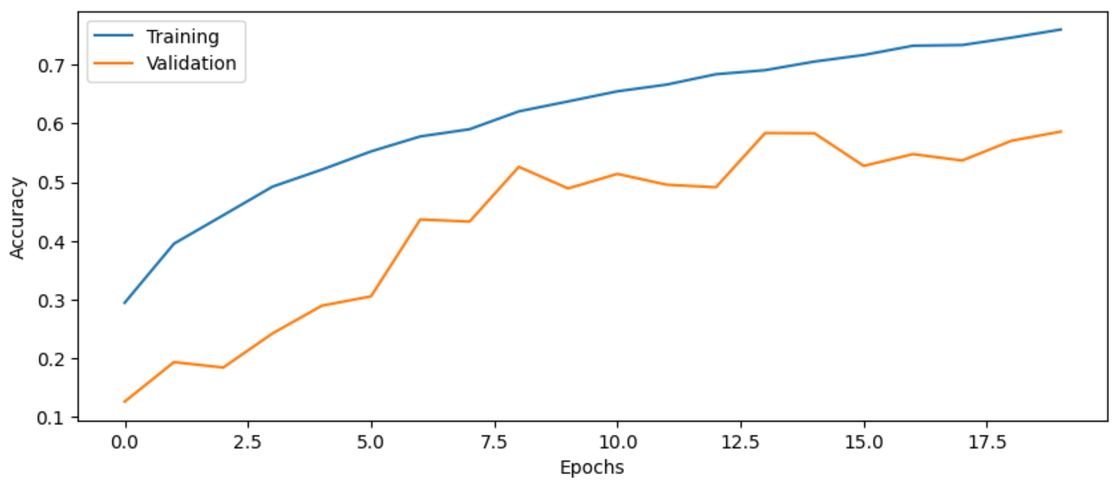
[59]:
```
# --------------------------------------------
# === Your code here =========================
# --------------------------------------------

# Setup some training parameters
batch_size = 128
epochs = 20
input_shape = Xtrain.shape[1:]
learning_rate = 0.01

# model architecture configurations
n_conv_layers = 2
n_dense_layers = 1
n_filters = 16
n_nodes = 32
use_dropout = True
optimizer = "adam"

# Build and train model. Here experiment with several model architecture␣
↪configurations to obtain the best performance.
model4 = build_CNN(input_shape=input_shape, loss="categorical_crossentropy",␣
↪n_conv_layers=n_conv_layers, n_dense_layers=n_dense_layers,␣
↪n_filters=n_filters, n_nodes=n_nodes, use_dropout=use_dropout,␣
↪optimizer=optimizer)

history4 = model4.fit(Xtrain, Ytrain, batch_size=batch_size, epochs=epochs,␣
↪validation_data=(Xval, Yval))

# Evaluate model on test data
score = history4.model.evaluate(Xtest, Ytest, batch_size=batch_size)
```

```python
# ===========================================

print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])

# Plot the history from the training run
plot_results(history4)
```

```
Epoch 1/20
59/59 [==============================] - 3s 18ms/step - loss: 2.0180 - accuracy:
0.2947 - val_loss: 2.7679 - val_accuracy: 0.1264
Epoch 2/20
59/59 [==============================] - 1s 11ms/step - loss: 1.6305 - accuracy:
0.3952 - val_loss: 2.6173 - val_accuracy: 0.1936
Epoch 3/20
59/59 [==============================] - 1s 11ms/step - loss: 1.5105 - accuracy:
0.4436 - val_loss: 2.6531 - val_accuracy: 0.1844
Epoch 4/20
59/59 [==============================] - 1s 11ms/step - loss: 1.3968 - accuracy:
0.4923 - val_loss: 2.4465 - val_accuracy: 0.2424
Epoch 5/20
59/59 [==============================] - 1s 11ms/step - loss: 1.3131 - accuracy:
0.5212 - val_loss: 2.3008 - val_accuracy: 0.2896
Epoch 6/20
59/59 [==============================] - 1s 11ms/step - loss: 1.2332 - accuracy:
0.5523 - val_loss: 2.0824 - val_accuracy: 0.3056
Epoch 7/20
59/59 [==============================] - 1s 11ms/step - loss: 1.1669 - accuracy:
0.5777 - val_loss: 1.7631 - val_accuracy: 0.4364
Epoch 8/20
59/59 [==============================] - 1s 12ms/step - loss: 1.1361 - accuracy:
0.5900 - val_loss: 1.9140 - val_accuracy: 0.4328
Epoch 9/20
59/59 [==============================] - 1s 12ms/step - loss: 1.0559 - accuracy:
0.6204 - val_loss: 1.4183 - val_accuracy: 0.5260
Epoch 10/20
59/59 [==============================] - 1s 12ms/step - loss: 1.0053 - accuracy:
0.6373 - val_loss: 1.6403 - val_accuracy: 0.4892
Epoch 11/20
59/59 [==============================] - 1s 12ms/step - loss: 0.9686 - accuracy:
0.6545 - val_loss: 1.4922 - val_accuracy: 0.5140
Epoch 12/20
59/59 [==============================] - 1s 12ms/step - loss: 0.9631 - accuracy:
0.6660 - val_loss: 1.7042 - val_accuracy: 0.4956
Epoch 13/20
59/59 [==============================] - 1s 12ms/step - loss: 0.8885 - accuracy:
0.6836 - val_loss: 1.7454 - val_accuracy: 0.4912
```

```
Epoch 14/20
59/59 [==============================] - 1s 11ms/step - loss: 0.8695 - accuracy:
0.6905 - val_loss: 1.3063 - val_accuracy: 0.5836
Epoch 15/20
59/59 [==============================] - 1s 12ms/step - loss: 0.8092 - accuracy:
0.7052 - val_loss: 1.2953 - val_accuracy: 0.5832
Epoch 16/20
59/59 [==============================] - 1s 12ms/step - loss: 0.7892 - accuracy:
0.7164 - val_loss: 1.5387 - val_accuracy: 0.5276
Epoch 17/20
59/59 [==============================] - 1s 12ms/step - loss: 0.7631 - accuracy:
0.7321 - val_loss: 1.4132 - val_accuracy: 0.5476
Epoch 18/20
59/59 [==============================] - 1s 11ms/step - loss: 0.7193 - accuracy:
0.7333 - val_loss: 1.5651 - val_accuracy: 0.5368
Epoch 19/20
59/59 [==============================] - 1s 12ms/step - loss: 0.7109 - accuracy:
0.7459 - val_loss: 1.4518 - val_accuracy: 0.5704
Epoch 20/20
59/59 [==============================] - 1s 12ms/step - loss: 0.6612 - accuracy:
0.7597 - val_loss: 1.3918 - val_accuracy: 0.5860
16/16 [==============================] - 0s 6ms/step - loss: 1.3593 - accuracy:
0.5795
Test loss: 1.3593
Test accuracy: 0.5795
```





## 6 Part 5: Model generalization

How high is the test accuracy if we rotate the test images? In other words, how good is the CNN at generalizing to rotated images?

Rotate each test image 90 degrees, the cells are already finished.

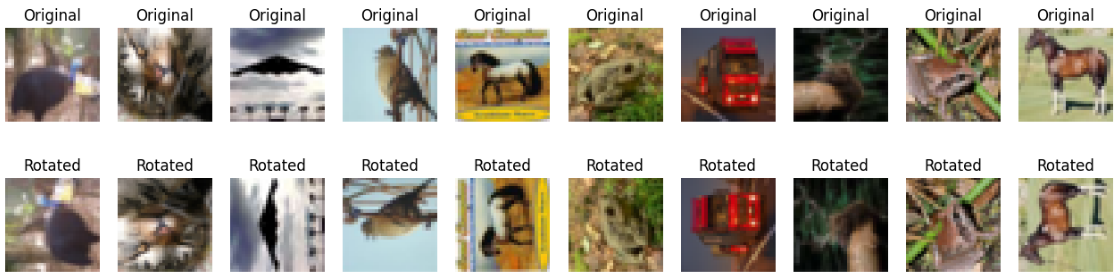
### Questions

24. What is the test accuracy for rotated test images, compared to test images without rotation? Explain the difference in accuracy.

### Answers

24. For the images without rotation we achieved a test accuracy of 0.5795. The test accuracy on the rotated images is significantly lower with 0.2480. The model was trained on the images without rotation and is not optimized to generalize on other images.

[60]:
```
from utilities import myrotate
# Visualize some rotated images
# Rotate the test images 90 degrees
Xtest_rotated = myrotate(Xtest)

# Look at some rotated images
plt.figure(figsize=(16,4))
for i in range(10):
    idx = np.random.randint(500)

    plt.subplot(2,10,i+1)
    plt.imshow(Xtest[idx]/2+0.5)
    plt.title("Original")
    plt.axis('off')

    plt.subplot(2,10,i+11)
    plt.imshow(Xtest_rotated[idx]/2+0.5)
    plt.title("Rotated")
    plt.axis('off')
plt.show()
```



[61]:
```
# Evaluate the trained model on rotated test set
score = model4.evaluate(Xtest_rotated, Ytest, verbose=0)
print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])
```

```
Test loss: 3.4857
Test accuracy: 0.2480
```

## 6.1 5.1 Augmentation using Keras ImageDataGenerator

We can increase the number of training images through data augmentation (we now ignore that CIFAR10 actually has 60 000 training images). Image augmentation is about creating similar images, by performing operations such as rotation, scaling, elastic deformations and flipping of existing images. This will prevent overfitting, especially if all the training images are in a certain orientation.

We will perform the augmentation on the fly, using a built-in function in Keras, called `ImageDataGenerator`. In particular, we will use the `flow()` functionality (see the documentation for more details).

Make sure to use different subsets for training and validation when you calling `flow()` on the training data generator in `model.fit()`, otherwise you will validate on the same data.

[62]:
```
# Get all 60 000 training images again. ImageDataGenerator manages validation␣
↪data on its own

# re-load the CIFAR10 train and test data
(X, Y), (Xtest, Ytest) = cifar10.load_data()
```

```
# Reduce the number of images for training/validation and testing to 10000 and␣
↪2000 respectively,
# to reduce processing time for this elaboration.
X = X[0:10000]
Y = Y[0:10000]

Xtest = Xtest[0:2000]
Ytest = Ytest[0:2000]

# Change data type and rescale range
X = X.astype('float32')
Xtest = Xtest.astype('float32')

X = X / 127.5 - 1
Xtest = Xtest / 127.5 - 1


# Convert labels to hot encoding
Y = to_categorical(Y, 10)
Ytest = to_categorical(Ytest, 10)

print("Training/validation images have size {} and labels have size {} ".
↪format(X.shape, Y.shape))
print("Test images have size {} and labels have size {} \n ".format(Xtest.
↪shape, Ytest.shape))
```

```
Training/validation images have size (10000, 32, 32, 3) and labels have size
(10000, 10)
Test images have size (2000, 32, 32, 3) and labels have size (2000, 10)
```

[64]:
```
from sklearn.model_selection import train_test_split

# --------------------------------------------
# === Your code here =========================
# --------------------------------------------

# split the original dataset into 75% Training and 25% Validation
Xtrain, Xval, Ytrain, Yval = train_test_split(X, Y, test_size=0.25,␣
↪random_state=42)



# Print the size of training data, validation data and test data
print("Training images have shape {} and labels have shape {} ".format(Xtrain.
↪shape, Ytrain.shape))
```

```
print("Validation images have shape {} and labels have shape {} ".format(Xval.
↪shape, Yval.shape))
print("Test images have shape {} and labels have shape {} ".format(Xtest.shape,␣
↪Ytest.shape))
# =============================================
```

```
Training images have shape (7500, 32, 32, 3) and labels have shape (7500, 10)
Validation images have shape (2500, 32, 32, 3) and labels have shape (2500, 10)
Test images have shape (2000, 32, 32, 3) and labels have shape (2000, 10)
```

[65]:
```
from tf_keras.preprocessing.image import ImageDataGenerator

# ---------------------------------------------
# === Your code here =========================
# ---------------------------------------------

# Use a rotation range of 30 degrees, horizontal and vertical flipping
# Set up image data generator
image_dataset = ImageDataGenerator(rotation_range=30, horizontal_flip=True,␣
↪vertical_flip=True)
train_flow = image_dataset.flow(Xtrain, Ytrain, batch_size=32)

# =============================================
```

### Questions

25. How would you change the code for the image generator if you cannot fit all training images in CPU memory? What is the disadvantage of doing that change?

### Answers

25. The flow function has a batch_size argument that can be adjusted. If this batch size is smaller than the used batch size in the training, it slows down the process.

[66]:
```
# Plot some augmented images

plt.figure(figsize=(12,4))
for i in range(18):
    (im, label) = train_flow.next()
    im = (im[0] + 1) * 127.5
    im = im.astype('int')
    label = np.flatnonzero(label)[0]

    plt.subplot(3,6,i+1)
    plt.tight_layout()
    plt.imshow(im)
    plt.title("Class: {} ({})".format(label, classes[label]))
    plt.axis('off')
plt.show()
```

## 6.2 5.2 Train the CNN with images from the generator

Check the documentation for the `model.fit` method how to use it with a generator instead of a fix dataset (numpy arrays).

To make the comparison fair to training without augmentation

- `steps_per_epoch` should be set to: `len(Xtrain)/batch_size`
- `validation_steps` should be set to: `len(Xval)/batch_size`

This is required since with a generator, the fit function will not know how many examples your original dataset has.

### Questions

26. How quickly is the training accuracy increasing compared to without augmentation? Explain why there is a difference compared to without augmentation. We are here talking about the number of training epochs required to reach a certain accuracy, and not the training time in seconds. What parameter is necessary to change to perform more training?
27. What other types of image augmentation can be applied, compared to what we use here?

### Answers

26. The training accuracy increases slower when using augmentation. With augmentation the variablility in the data. It helps the model to generalize better but also makes it harder to predict correctly in the beginning. As the low difference between shows, the overfitting is mitigated. To perform more training we could increase the amount of epochs.
27. The ImageDataGenerator offers many more possibile augmentation techniques:
    - width or height shift: positional change along an axis
    - adjust brightness
    - shear transformation: skew image along an axis
    - zoom
    - channel shift: adjust colors

[68]:
```python
# ---------------------------------------------
# === Your code here ========================
# ---------------------------------------------

# Setup some training parameters
batch_size = 128
epochs = 20
input_shape = Xtrain.shape[1:]
learning_rate = 0.01

# model architecture configurations
n_conv_layers = 2
n_dense_layers = 1
n_filters = 16
n_nodes = 32
use_dropout = True
optimizer = "adam"

# Build model (your best config)
model6 = build_CNN(input_shape=input_shape, loss="categorical_crossentropy",␣
 ↪n_conv_layers=n_conv_layers, n_dense_layers=n_dense_layers,␣
 ↪n_filters=n_filters, n_nodes=n_nodes, use_dropout=use_dropout,␣
 ↪optimizer=optimizer)


# Set up training and validation dataset flows from image_dataset
# flow() for training data
train_flow = image_dataset.flow(Xtrain, Ytrain, batch_size=128)

# flow() for validation data
val_flow = image_dataset.flow(Xval, Yval, batch_size=128)


# Train the model using on the fly augmentation
history6 = model6.fit(train_flow, batch_size=batch_size, epochs=epochs,␣
 ↪validation_data=val_flow, steps_per_epoch=len(Xtrain)/batch_size,␣
 ↪validation_steps=len(Xval)/batch_size)

# =============================================
```

```
Epoch 1/20
58/58 [==============================] - 5s 43ms/step - loss: 2.1725 - accuracy:
0.2443 - val_loss: 3.1713 - val_accuracy: 0.1044
Epoch 2/20
58/58 [==============================] - 2s 39ms/step - loss: 1.8576 - accuracy:
0.3179 - val_loss: 3.4237 - val_accuracy: 0.1248
Epoch 3/20
58/58 [==============================] - 2s 39ms/step - loss: 1.7421 - accuracy:
```

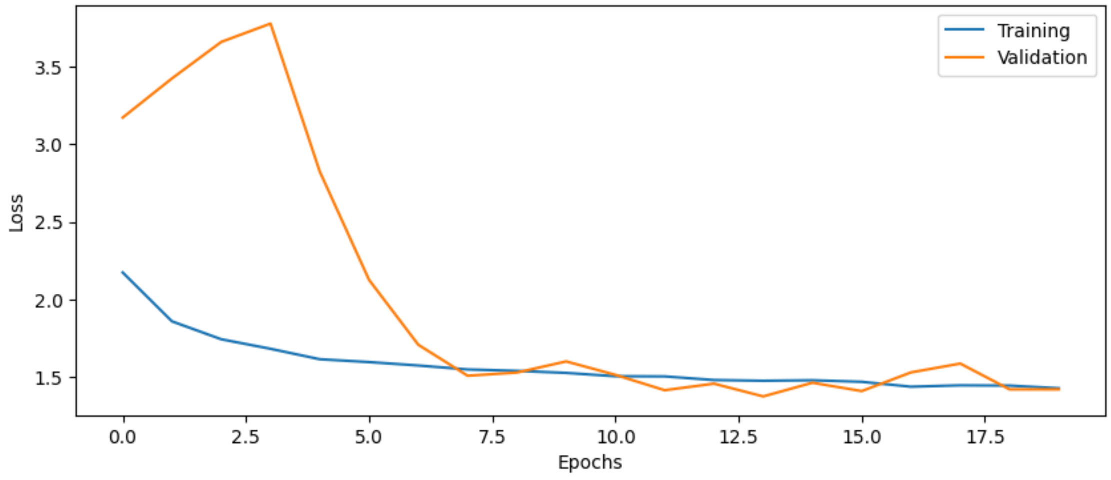
```
0.3529 - val_loss: 3.6590 - val_accuracy: 0.1504
Epoch 4/20
58/58 [==============================] - 2s 39ms/step - loss: 1.6807 - accuracy:
0.3688 - val_loss: 3.7772 - val_accuracy: 0.1492
Epoch 5/20
58/58 [==============================] - 2s 39ms/step - loss: 1.6133 - accuracy:
0.3907 - val_loss: 2.8227 - val_accuracy: 0.1772
Epoch 6/20
58/58 [==============================] - 2s 41ms/step - loss: 1.5949 - accuracy:
0.4020 - val_loss: 2.1248 - val_accuracy: 0.2916
Epoch 7/20
58/58 [==============================] - 2s 39ms/step - loss: 1.5729 - accuracy:
0.4175 - val_loss: 1.7064 - val_accuracy: 0.3704
Epoch 8/20
58/58 [==============================] - 2s 38ms/step - loss: 1.5475 - accuracy:
0.4212 - val_loss: 1.5070 - val_accuracy: 0.4456
Epoch 9/20
58/58 [==============================] - 2s 40ms/step - loss: 1.5384 - accuracy:
0.4271 - val_loss: 1.5274 - val_accuracy: 0.4168
Epoch 10/20
58/58 [==============================] - 2s 39ms/step - loss: 1.5252 - accuracy:
0.4324 - val_loss: 1.5992 - val_accuracy: 0.3756
Epoch 11/20
58/58 [==============================] - 2s 39ms/step - loss: 1.5037 - accuracy:
0.4365 - val_loss: 1.5126 - val_accuracy: 0.4524
Epoch 12/20
58/58 [==============================] - 2s 38ms/step - loss: 1.5026 - accuracy:
0.4487 - val_loss: 1.4136 - val_accuracy: 0.4784
Epoch 13/20
58/58 [==============================] - 2s 40ms/step - loss: 1.4794 - accuracy:
0.4523 - val_loss: 1.4558 - val_accuracy: 0.4604
Epoch 14/20
58/58 [==============================] - 2s 39ms/step - loss: 1.4744 - accuracy:
0.4517 - val_loss: 1.3736 - val_accuracy: 0.5092
Epoch 15/20
58/58 [==============================] - 2s 39ms/step - loss: 1.4777 - accuracy:
0.4535 - val_loss: 1.4618 - val_accuracy: 0.4672
Epoch 16/20
58/58 [==============================] - 2s 38ms/step - loss: 1.4678 - accuracy:
0.4603 - val_loss: 1.4079 - val_accuracy: 0.4776
Epoch 17/20
58/58 [==============================] - 2s 39ms/step - loss: 1.4363 - accuracy:
0.4663 - val_loss: 1.5284 - val_accuracy: 0.4460
Epoch 18/20
58/58 [==============================] - 2s 38ms/step - loss: 1.4451 - accuracy:
0.4621 - val_loss: 1.5854 - val_accuracy: 0.4236
Epoch 19/20
58/58 [==============================] - 2s 40ms/step - loss: 1.4435 - accuracy:
```

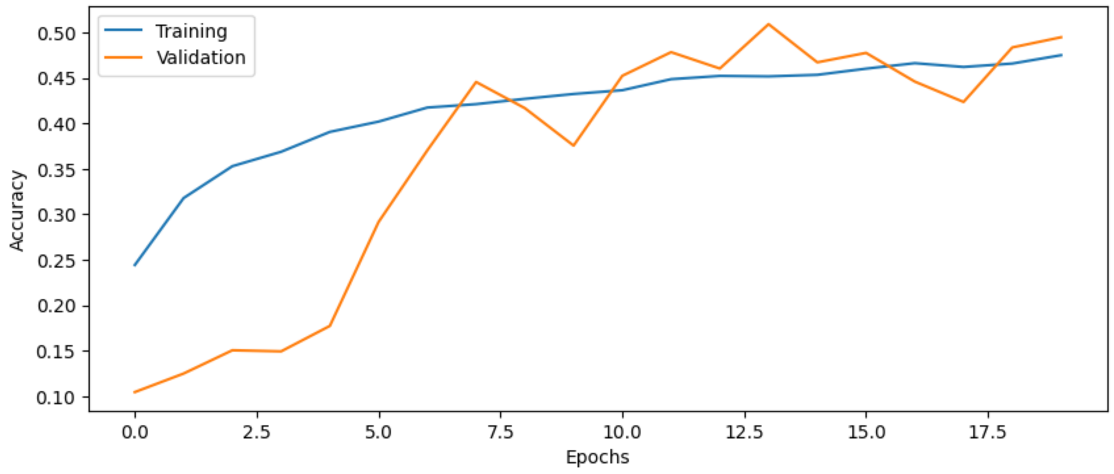
```
0.4659 - val_loss: 1.4191 - val_accuracy: 0.4836
Epoch 20/20
58/58 [==============================] - 2s 40ms/step - loss: 1.4262 - accuracy:
0.4751 - val_loss: 1.4191 - val_accuracy: 0.4948
```

[69]:
```
# Check if there is still a big difference in accuracy for original and rotated␣
↪test images

# Evaluate the trained model on original test set
score = model6.evaluate(Xtest, Ytest, batch_size = batch_size, verbose=0)
print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])

# Evaluate the trained model on rotated test set
score = model6.evaluate(Xtest_rotated, Ytest, batch_size = batch_size,␣
↪verbose=0)
print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])
```

```
Test loss: 1.4760
Test accuracy: 0.4910
Test loss: 2.1859
Test accuracy: 0.3115
```

[70]:
```
# Plot the history from the training run
plot_results(history6)
```





### 6.3 Plot misclassified images

Lets plot some images where the CNN performed badly.

[71]:
```
# Find misclassified images
y_pred=model6.predict(Xtest, verbose=0)
y_pred=np.argmax(y_pred,axis=1)

y_correct = np.argmax(Ytest,axis=-1)

miss = np.flatnonzero(y_correct != y_pred)
```

[72]:
```
# Plot a few of them
plt.figure(figsize=(15,4))
perm = np.random.permutation(miss)
for i in range(18):
    im = (Xtest[perm[i]] + 1) * 127.5
    im = im.astype('int')
    label_correct = y_correct[perm[i]]
    label_pred = y_pred[perm[i]]

    plt.subplot(3,6,i+1)
    plt.tight_layout()
    plt.imshow(im)
    plt.axis('off')
    plt.title("{}, classified as {}".format(classes[label_correct],␣
 ↪classes[label_pred]))
plt.show()
```

## 6.4 5.3 Testing on another size

### Questions

28. This CNN has been trained on 32 x 32 images, can it be applied to images of another size? If not, why is this the case?
29. Is it possible to design a CNN that can be trained on images of one size, and then applied to an image of any size? How?

### Answers

28. The model can't be applied to images of another size, because the input layer expects a certain input size that was defined in the build_CNN() function. Additionally, the dense layers expect a vector of a certain length from the flatten layer. It would be different for other image sizes.
29. Fully Convolutional networks replace the dense layers with convolutional layers. The latter are not dependent on the input size. Alternatively, the flatten layer could be replaced with a layer that performs global average pooling, which results in a vector with fixed size.

# 7 Part 6: Carbon footprint

In this next section we will evaluate the carbon footprint of training our CNN model. In particular we will look at the effect of training hyper parameters of carbon footprint. You can read more about this topic here or here.

In this lab we will use the `carbontracker` library that easily integrates with any model training routine. See the example in the documentation on how to use the carbon tracker.

### Questions

28. Keeping the model architecture fixed, which training parameter impacts the carbon footprint?
29. The choice of batch size can dramatically impact carbon foot print: why is this the case?
30. Assume that you have a model with 100 million parameters running in the backend of a service with 5 million users. How can the carbon footprint of using this model be reduced?

**Answers**

28. The amount of epochs impacts the carbon footprint as it decides how long the training runs. The batch size has also an impact, because the amount of steps per epoch depend on that training parameter.
29. A smaller batch size causes steps with backpropagation and therefore impacts the carbon footprint.
30. Running the backend with power from renewable sources can lower the carbon footprint. With this approach we would not have to adjust the model or limit the access for the users.

[15]:
```
from utilities import build_CNN, plot_results
```

[ ]:
```
from carbontracker.tracker import CarbonTracker

# --------------------------------------------
# === Your code here =========================
# --------------------------------------------
# Setup some training parameters
batch_size = 128
epochs = 20
input_shape = Xtrain.shape[1:]
learning_rate = 0.01

# model architecture configurations
n_conv_layers = 2
n_dense_layers = 1
n_filters = 16
n_nodes = 32
use_dropout = True
optimizer = "adam"

# Build and train model. Here experiment with several model architecture␣
 ↪configurations to obtain the best performance.
model7 = build_CNN(input_shape=input_shape, loss="categorical_crossentropy",␣
 ↪n_conv_layers=n_conv_layers, n_dense_layers=n_dense_layers,␣
 ↪n_filters=n_filters, n_nodes=n_nodes, use_dropout=use_dropout,␣
 ↪optimizer=optimizer)


# Create a CarbonTracker object
tracker = CarbonTracker(epochs=epochs)

# start carbon tracking
tracker.epoch_start()

# fit model
```

```
model7.fit(Xtrain, Ytrain, batch_size=batch_size, epochs=epochs,␣
 ↪validation_data=(Xval, Yval))

tracker.epoch_end()

# ============================================
```

```
I0000 00:00:1741597082.016709    4179 gpu_device.cc:2022] Created device
/job:localhost/replica:0/task:0/device:GPU:0 with 5563 MB memory:  -> device: 0,
name: NVIDIA GeForce RTX 4060 Laptop GPU, pci bus id: 0000:01:00.0, compute
capability: 8.9
```

```
CarbonTracker: The following components were found: GPU with device(s) NVIDIA
GeForce RTX 4060 Laptop GPU.
CarbonTracker: WARNING - ElectricityMaps API key not set. Will default to
average carbon intensity.
CarbonTracker: WARNING - Failed to retrieve carbon intensity: Defaulting to
average carbon intensity 40.694878 gCO2/kWh.
Epoch 1/20
```

```
I0000 00:00:1741597084.638851    4399 cuda_dnn.cc:529] Loaded cuDNN version
90300
WARNING: All log messages before absl::InitializeLog() is called are written to
STDERR
I0000 00:00:1741597086.962610    4403 service.cc:148] XLA service 0x7f5d09cec780
initialized for platform CUDA (this does not guarantee that XLA will be used).
Devices:
I0000 00:00:1741597086.963185    4403 service.cc:156]   StreamExecutor device
(0): NVIDIA GeForce RTX 4060 Laptop GPU, Compute Capability 8.9
2025-03-10 09:58:06.979687: I
tensorflow/compiler/mlir/tensorflow/utils/dump_mlir_util.cc:268] disabling MLIR
crash reproducer, set env var `MLIR_CRASH_REPRODUCER_DIRECTORY` to enable.
I0000 00:00:1741597087.140141    4403 device_compiler.h:188] Compiled cluster
using XLA!  This line is logged at most once for the lifetime of the process.
```

```
59/59 [==============================] - 7s 26ms/step - loss: 2.0836 - accuracy:
0.2708 - val_loss: 3.0426 - val_accuracy: 0.1064
Epoch 2/20
59/59 [==============================] - 1s 12ms/step - loss: 1.6951 - accuracy:
0.3753 - val_loss: 3.1292 - val_accuracy: 0.1548
Epoch 3/20
59/59 [==============================] - 1s 13ms/step - loss: 1.5361 - accuracy:
0.4317 - val_loss: 2.7987 - val_accuracy: 0.1444
Epoch 4/20
59/59 [==============================] - 1s 13ms/step - loss: 1.4647 - accuracy:
0.4684 - val_loss: 2.7250 - val_accuracy: 0.1712
Epoch 5/20
59/59 [==============================] - 1s 14ms/step - loss: 1.3925 - accuracy:
0.4993 - val_loss: 2.6491 - val_accuracy: 0.2432
```

```
Epoch 6/20
59/59 [==============================] - 1s 13ms/step - loss: 1.3024 - accuracy:
0.5251 - val_loss: 2.1093 - val_accuracy: 0.3516
Epoch 7/20
59/59 [==============================] - 1s 13ms/step - loss: 1.2625 - accuracy:
0.5400 - val_loss: 1.8821 - val_accuracy: 0.3740
Epoch 8/20
59/59 [==============================] - 1s 13ms/step - loss: 1.1880 - accuracy:
0.5691 - val_loss: 1.6751 - val_accuracy: 0.4568
Epoch 9/20
59/59 [==============================] - 1s 13ms/step - loss: 1.1571 - accuracy:
0.5809 - val_loss: 1.5383 - val_accuracy: 0.4852
Epoch 10/20
59/59 [==============================] - 1s 16ms/step - loss: 1.0956 - accuracy:
0.6056 - val_loss: 2.3538 - val_accuracy: 0.3792
Epoch 11/20
59/59 [==============================] - 1s 14ms/step - loss: 1.0922 - accuracy:
0.6055 - val_loss: 1.4537 - val_accuracy: 0.5184
Epoch 12/20
59/59 [==============================] - 1s 16ms/step - loss: 0.9951 - accuracy:
0.6393 - val_loss: 1.3548 - val_accuracy: 0.5616
Epoch 13/20
59/59 [==============================] - 1s 12ms/step - loss: 0.9603 - accuracy:
0.6540 - val_loss: 1.3744 - val_accuracy: 0.5508
Epoch 14/20
59/59 [==============================] - 1s 14ms/step - loss: 0.9313 - accuracy:
0.6692 - val_loss: 1.2178 - val_accuracy: 0.5932
Epoch 15/20
59/59 [==============================] - 1s 13ms/step - loss: 0.9007 - accuracy:
0.6720 - val_loss: 1.3257 - val_accuracy: 0.5668
Epoch 16/20
59/59 [==============================] - 0s 156us/step - loss: 0.8394 -
accuracy: 0.7015 - val_loss: 1.3105 - val_accuracy: 0.5648
Epoch 17/20
59/59 [==============================] - 1s 18ms/step - loss: 0.8459 - accuracy:
0.6972 - val_loss: 1.3052 - val_accuracy: 0.5744
Epoch 18/20
59/59 [==============================] - 1s 16ms/step - loss: 0.7839 - accuracy:
0.7172 - val_loss: 1.3779 - val_accuracy: 0.5788
Epoch 19/20
59/59 [==============================] - 1s 17ms/step - loss: 0.7857 - accuracy:
0.7173 - val_loss: 1.3209 - val_accuracy: 0.5796
Epoch 20/20
59/59 [==============================] - 1s 15ms/step - loss: 0.7337 - accuracy:
0.7416 - val_loss: 1.4132 - val_accuracy: 0.5624
CarbonTracker: WARNING - ElectricityMaps API key not set. Will default to
average carbon intensity.
CarbonTracker: WARNING - Failed to retrieve carbon intensity: Defaulting to
```

```
average carbon intensity 40.694878 gCO2/kWh.
CarbonTracker: Live carbon intensity could not be fetched at detected location:
Stockholm, Stockholm, SE. Defaulted to average carbon intensity for SE in 2023
of 40.69 gCO2/kWh. at detected location: Stockholm, Stockholm, SE.
CarbonTracker:
Predicted consumption for 20 epoch(s):
        Time:   0:07:26
        Energy: 0.003208444148 kWh
        CO2eq:  0.130567243159 g
        This is equivalent to:
        0.001214579006 km travelled by car

CarbonTracker: WARNING - ElectricityMaps API key not set. Will default to
average carbon intensity.
CarbonTracker: WARNING - Failed to retrieve carbon intensity: Defaulting to
average carbon intensity 40.694878 gCO2/kWh.
```

[17]:
```
epochs = 10
# Create a CarbonTracker object
tracker = CarbonTracker(epochs=epochs)

# start carbon tracking
tracker.epoch_start()

# fit model
model7.fit(Xtrain, Ytrain, batch_size=batch_size, epochs=epochs,␣
 ↪validation_data=(Xval, Yval))

tracker.epoch_end()
```

```
CarbonTracker: WARNING - ElectricityMaps API key not set. Will default to
average carbon intensity.
CarbonTracker: WARNING - Failed to retrieve carbon intensity: Defaulting to
average carbon intensity 40.694878 gCO2/kWh.
Epoch 1/10
59/59 [==============================] - 1s 22ms/step - loss: 0.7353 - accuracy:
0.7357 - val_loss: 1.2608 - val_accuracy: 0.5976
Epoch 2/10
59/59 [==============================] - 1s 17ms/step - loss: 0.6994 - accuracy:
0.7460 - val_loss: 1.4404 - val_accuracy: 0.5560
Epoch 3/10
59/59 [==============================] - 1s 15ms/step - loss: 0.6777 - accuracy:
0.7603 - val_loss: 1.5705 - val_accuracy: 0.5456
Epoch 4/10
59/59 [==============================] - 1s 14ms/step - loss: 0.6436 - accuracy:
0.7647 - val_loss: 1.4284 - val_accuracy: 0.5636
Epoch 5/10
59/59 [==============================] - 1s 15ms/step - loss: 0.6450 - accuracy:
```

```
0.7691 - val_loss: 1.4364 - val_accuracy: 0.5808
Epoch 6/10
59/59 [==============================] - 1s 14ms/step - loss: 0.5994 - accuracy:
0.7824 - val_loss: 1.4933 - val_accuracy: 0.5788
Epoch 7/10
59/59 [==============================] - 1s 15ms/step - loss: 0.5991 - accuracy:
0.7895 - val_loss: 1.5035 - val_accuracy: 0.5740
Epoch 8/10
59/59 [==============================] - 1s 16ms/step - loss: 0.6052 - accuracy:
0.7829 - val_loss: 1.5628 - val_accuracy: 0.5576
Epoch 9/10
59/59 [==============================] - 1s 16ms/step - loss: 0.5578 - accuracy:
0.7967 - val_loss: 1.4639 - val_accuracy: 0.5908
Epoch 10/10
59/59 [==============================] - 1s 14ms/step - loss: 0.5434 - accuracy:
0.8009 - val_loss: 1.4791 - val_accuracy: 0.5764
CarbonTracker: WARNING - ElectricityMaps API key not set. Will default to
average carbon intensity.
CarbonTracker: WARNING - Failed to retrieve carbon intensity: Defaulting to
average carbon intensity 40.694878 gCO2/kWh.
CarbonTracker: Live carbon intensity could not be fetched at detected location:
Stockholm, Stockholm, SE. Defaulted to average carbon intensity for SE in 2023
of 40.69 gCO2/kWh. at detected location: Stockholm, Stockholm, SE.
CarbonTracker:
Predicted consumption for 10 epoch(s):
        Time:   0:01:44
        Energy: 0.000616445259 kWh
        CO2eq:  0.025086164604 g
        This is equivalent to:
        0.000233359671 km travelled by car
```

[18]:
```python
epochs = 10
batch_size = 64

# Create a CarbonTracker object
tracker = CarbonTracker(epochs=epochs)

# start carbon tracking
tracker.epoch_start()

# fit model
model7.fit(Xtrain, Ytrain, batch_size=batch_size, epochs=epochs,␣
 ↪validation_data=(Xval, Yval))

tracker.epoch_end()
```

```
CarbonTracker: WARNING - ElectricityMaps API key not set. Will default to
average carbon intensity.
```

```
CarbonTracker: WARNING - Failed to retrieve carbon intensity: Defaulting to
average carbon intensity 40.694878 gCO2/kWh.
Epoch 1/10
118/118 [==============================] - 3s 19ms/step - loss: 0.7880 -
accuracy: 0.7253 - val_loss: 1.4729 - val_accuracy: 0.5572
Epoch 2/10
118/118 [==============================] - 2s 13ms/step - loss: 0.7395 -
accuracy: 0.7349 - val_loss: 1.5811 - val_accuracy: 0.5536
Epoch 3/10
118/118 [==============================] - 2s 13ms/step - loss: 0.7251 -
accuracy: 0.7412 - val_loss: 1.4579 - val_accuracy: 0.5624
Epoch 4/10
118/118 [==============================] - 2s 14ms/step - loss: 0.6924 -
accuracy: 0.7441 - val_loss: 1.4641 - val_accuracy: 0.5800
Epoch 5/10
118/118 [==============================] - 2s 14ms/step - loss: 0.6517 -
accuracy: 0.7652 - val_loss: 1.4961 - val_accuracy: 0.5568
Epoch 6/10
118/118 [==============================] - 2s 14ms/step - loss: 0.6718 -
accuracy: 0.7612 - val_loss: 1.5232 - val_accuracy: 0.5476
Epoch 7/10
118/118 [==============================] - 2s 15ms/step - loss: 0.7558 -
accuracy: 0.7309 - val_loss: 1.3770 - val_accuracy: 0.5804
Epoch 8/10
118/118 [==============================] - 2s 14ms/step - loss: 0.6085 -
accuracy: 0.7843 - val_loss: 1.4892 - val_accuracy: 0.5620
Epoch 9/10
118/118 [==============================] - 1s 8ms/step - loss: 0.5961 -
accuracy: 0.7884 - val_loss: 1.5053 - val_accuracy: 0.5544
Epoch 10/10
118/118 [==============================] - 2s 14ms/step - loss: 0.5573 -
accuracy: 0.8008 - val_loss: 1.5185 - val_accuracy: 0.5604
CarbonTracker: WARNING - ElectricityMaps API key not set. Will default to
average carbon intensity.
CarbonTracker: WARNING - Failed to retrieve carbon intensity: Defaulting to
average carbon intensity 40.694878 gCO2/kWh.
CarbonTracker: Live carbon intensity could not be fetched at detected location:
Stockholm, Stockholm, SE. Defaulted to average carbon intensity for SE in 2023
of 40.69 gCO2/kWh. at detected location: Stockholm, Stockholm, SE.
CarbonTracker:
Predicted consumption for 10 epoch(s):
        Time:   0:02:52
        Energy: 0.001024363020 kWh
        CO2eq:  0.041686328140 g
        This is equivalent to:
        0.000387779797 km travelled by car

CarbonTracker: WARNING - ElectricityMaps API key not set. Will default to
```

```
average carbon intensity.
CarbonTracker: WARNING - Failed to retrieve carbon intensity: Defaulting to
average carbon intensity 40.694878 gCO2/kWh.
```

## 8 Part 7: Pre-trained 2D CNNs

There are many deep 2D CNNs that have been pre-trained using the large ImageNet database (several million images, 1000 classes). Import a pre-trained ResNet50 network from Keras applications. Show the network using `model.summary()`

### Questions

31. How many convolutional layers does ResNet50 have?
32. How many trainable parameters does the ResNet50 network have?
33. What is the size of the images that ResNet50 expects as input?
34. Using the answer to question 30, explain why the second derivative is seldom used when training deep networks.
35. What do you expect the carbon footprint of using pre-trained networks to be compared to training a model from scratch?

### Answers

31. It has 49 convolutional layers.
32. It has 25,583,592 trainable parameters.
33. It expacts images with (224,224,3) input shape.
34. The computational cost to compute the second derivative is too big for larger models, as the hessian matrix must be computed every time.
35. It should be lower, because the pre-trained model was once trained for many users and simply provides the weights, which does not cause any new carbon emissions. Training a similarly powerful model from scratch is way mroe expensive and therfore has the bigger carbon footprint.

After loading the pre-trained CNN, apply it to 5 random color images that you download and copy to the cloud machine or your own computer. Are the predictions correct? How certain is the network of each image class?

These pre-trained networks can be fine tuned to your specific data, and normally only the last layers need to be re-trained, but it will still be too time consuming to do in this elaboration.

Some useful functions: - `load_img` and `img_to_array` in tf_keras.utils. - `ResNet50` in tf_keras.applications.ResNet50. - `preprocess_input` in tf_keras.applications.resnet. - `decode_predictions` in tf_keras.applications.resnet. - `expand_dims` in numpy.

See keras applications and the keras resnet50-function for more details.

[16]:
```python
# --------------------------------------------
# === Your code here ========================
# --------------------------------------------
# import the necessary libraries and functions
from tf_keras.applications import ResNet50
from tf_keras.utils import load_img, img_to_array
from tf_keras.applications.resnet import decode_predictions, preprocess_input
from tf_keras.preprocessing.image import smart_resize
from numpy import expand_dims

# load the pre-trained ResNet50 model
resnet50 = ResNet50(weights='imagenet')

# print the model summary
print(resnet50.summary())

for path in ["dog.png", "cat.png", "jet.png", "elpresidente.png", "empirestate.
 ↪png"]:
    # load the image and preprocess it
    image = load_img(path)
    image = img_to_array(image)
    image = smart_resize(image, (224, 224))
    image = expand_dims(image, axis=0)
    image = preprocess_input(image)

    # predict the image
    label = resnet50.predict(image)
    label = decode_predictions(label)
    # print the predicted label
    print(path)
    print(label)

# ============================================
```

```
Model: "resnet50"

________________________________________________________________________________
__________________
 Layer (type)                Output Shape                 Param #   Connected to
================================================================================
==================
 input_13 (InputLayer)       [(None, 224, 224, 3)]        0         []

 conv1_pad (ZeroPadding2D)   (None, 230, 230, 3)          0
['input_13[0][0]']

 conv1_conv (Conv2D)         (None, 112, 112, 64)         9472
['conv1_pad[0][0]']
```

```
 conv1_bn (BatchNormalizati  (None, 112, 112, 64)         256
['conv1_conv[0][0]']
 on)

 conv1_relu (Activation)     (None, 112, 112, 64)         0
['conv1_bn[0][0]']

 pool1_pad (ZeroPadding2D)   (None, 114, 114, 64)         0
['conv1_relu[0][0]']

 pool1_pool (MaxPooling2D)   (None, 56, 56, 64)           0
['pool1_pad[0][0]']

 conv2_block1_1_conv (Conv2  (None, 56, 56, 64)           4160
['pool1_pool[0][0]']
 D)

 conv2_block1_1_bn (BatchNo  (None, 56, 56, 64)           256
['conv2_block1_1_conv[0][0]']
 rmalization)

 conv2_block1_1_relu (Activ  (None, 56, 56, 64)           0
['conv2_block1_1_bn[0][0]']
 ation)

 conv2_block1_2_conv (Conv2  (None, 56, 56, 64)           36928
['conv2_block1_1_relu[0][0]']
 D)

 conv2_block1_2_bn (BatchNo  (None, 56, 56, 64)           256
['conv2_block1_2_conv[0][0]']
 rmalization)

 conv2_block1_2_relu (Activ  (None, 56, 56, 64)           0
['conv2_block1_2_bn[0][0]']
 ation)

 conv2_block1_0_conv (Conv2  (None, 56, 56, 256)          16640
['pool1_pool[0][0]']
 D)

 conv2_block1_3_conv (Conv2  (None, 56, 56, 256)          16640
['conv2_block1_2_relu[0][0]']
 D)

 conv2_block1_0_bn (BatchNo  (None, 56, 56, 256)          1024
['conv2_block1_0_conv[0][0]']
 rmalization)
 conv2_block1_3_bn (BatchNo  (None, 56, 56, 256)          1024
['conv2_block1_3_conv[0][0]']
 rmalization)

 conv2_block1_add (Add)      (None, 56, 56, 256)          0
['conv2_block1_0_bn[0][0]',
'conv2_block1_3_bn[0][0]']

 conv2_block1_out (Activati  (None, 56, 56, 256)          0
['conv2_block1_add[0][0]']
 on)

 conv2_block2_1_conv (Conv2  (None, 56, 56, 64)           16448
['conv2_block1_out[0][0]']
 D)

 conv2_block2_1_bn (BatchNo  (None, 56, 56, 64)           256
['conv2_block2_1_conv[0][0]']
 rmalization)

 conv2_block2_1_relu (Activ  (None, 56, 56, 64)           0
['conv2_block2_1_bn[0][0]']
 ation)

 conv2_block2_2_conv (Conv2  (None, 56, 56, 64)           36928
['conv2_block2_1_relu[0][0]']
 D)

 conv2_block2_2_bn (BatchNo  (None, 56, 56, 64)           256
['conv2_block2_2_conv[0][0]']
 rmalization)

 conv2_block2_2_relu (Activ  (None, 56, 56, 64)           0
['conv2_block2_2_bn[0][0]']
 ation)

 conv2_block2_3_conv (Conv2  (None, 56, 56, 256)          16640
['conv2_block2_2_relu[0][0]']
 D)

 conv2_block2_3_bn (BatchNo  (None, 56, 56, 256)          1024
['conv2_block2_3_conv[0][0]']
 rmalization)

 conv2_block2_add (Add)      (None, 56, 56, 256)          0
['conv2_block1_out[0][0]',
'conv2_block2_3_bn[0][0]']
```

```
 conv2_block2_out (Activati  (None, 56, 56, 256)          0
['conv2_block2_add[0][0]']
 on)

 conv2_block3_1_conv (Conv2  (None, 56, 56, 64)           16448
['conv2_block2_out[0][0]']
 D)

 conv2_block3_1_bn (BatchNo  (None, 56, 56, 64)           256
['conv2_block3_1_conv[0][0]']
 rmalization)

 conv2_block3_1_relu (Activ  (None, 56, 56, 64)           0
['conv2_block3_1_bn[0][0]']
 ation)

 conv2_block3_2_conv (Conv2  (None, 56, 56, 64)           36928
['conv2_block3_1_relu[0][0]']
 D)

 conv2_block3_2_bn (BatchNo  (None, 56, 56, 64)           256
['conv2_block3_2_conv[0][0]']
 rmalization)

 conv2_block3_2_relu (Activ  (None, 56, 56, 64)           0
['conv2_block3_2_bn[0][0]']
 ation)

 conv2_block3_3_conv (Conv2  (None, 56, 56, 256)          16640
['conv2_block3_2_relu[0][0]']
 D)

 conv2_block3_3_bn (BatchNo  (None, 56, 56, 256)          1024
['conv2_block3_3_conv[0][0]']
 rmalization)

 conv2_block3_add (Add)      (None, 56, 56, 256)          0
['conv2_block2_out[0][0]',
'conv2_block3_3_bn[0][0]']

 conv2_block3_out (Activati  (None, 56, 56, 256)          0
['conv2_block3_add[0][0]']
 on)

 conv3_block1_1_conv (Conv2  (None, 28, 28, 128)          32896
['conv2_block3_out[0][0]']
 D)
 conv3_block1_1_bn (BatchNo  (None, 28, 28, 128)          512
['conv3_block1_1_conv[0][0]']
 rmalization)

 conv3_block1_1_relu (Activ  (None, 28, 28, 128)          0
['conv3_block1_1_bn[0][0]']
 ation)

 conv3_block1_2_conv (Conv2  (None, 28, 28, 128)          147584
['conv3_block1_1_relu[0][0]']
 D)

 conv3_block1_2_bn (BatchNo  (None, 28, 28, 128)          512
['conv3_block1_2_conv[0][0]']
 rmalization)

 conv3_block1_2_relu (Activ  (None, 28, 28, 128)          0
['conv3_block1_2_bn[0][0]']
 ation)

 conv3_block1_0_conv (Conv2  (None, 28, 28, 512)          131584
['conv2_block3_out[0][0]']
 D)

 conv3_block1_3_conv (Conv2  (None, 28, 28, 512)          66048
['conv3_block1_2_relu[0][0]']
 D)

 conv3_block1_0_bn (BatchNo  (None, 28, 28, 512)          2048
['conv3_block1_0_conv[0][0]']
 rmalization)

 conv3_block1_3_bn (BatchNo  (None, 28, 28, 512)          2048
['conv3_block1_3_conv[0][0]']
 rmalization)

 conv3_block1_add (Add)      (None, 28, 28, 512)          0
['conv3_block1_0_bn[0][0]',
'conv3_block1_3_bn[0][0]']

 conv3_block1_out (Activati  (None, 28, 28, 512)          0
['conv3_block1_add[0][0]']
 on)

 conv3_block2_1_conv (Conv2  (None, 28, 28, 128)          65664
['conv3_block1_out[0][0]']
 D)
 conv3_block2_1_bn (BatchNo  (None, 28, 28, 128)          512
['conv3_block2_1_conv[0][0]']
 rmalization)

 conv3_block2_1_relu (Activ  (None, 28, 28, 128)          0
['conv3_block2_1_bn[0][0]']
 ation)

 conv3_block2_2_conv (Conv2  (None, 28, 28, 128)          147584
['conv3_block2_1_relu[0][0]']
 D)

 conv3_block2_2_bn (BatchNo  (None, 28, 28, 128)          512
['conv3_block2_2_conv[0][0]']
 rmalization)

 conv3_block2_2_relu (Activ  (None, 28, 28, 128)          0
['conv3_block2_2_bn[0][0]']
 ation)

 conv3_block2_3_conv (Conv2  (None, 28, 28, 512)          66048
['conv3_block2_2_relu[0][0]']
 D)

 conv3_block2_3_bn (BatchNo  (None, 28, 28, 512)          2048
['conv3_block2_3_conv[0][0]']
 rmalization)

 conv3_block2_add (Add)      (None, 28, 28, 512)          0
['conv3_block1_out[0][0]',
'conv3_block2_3_bn[0][0]']

 conv3_block2_out (Activati  (None, 28, 28, 512)          0
['conv3_block2_add[0][0]']
 on)

 conv3_block3_1_conv (Conv2  (None, 28, 28, 128)          65664
['conv3_block2_out[0][0]']
 D)

 conv3_block3_1_bn (BatchNo  (None, 28, 28, 128)          512
['conv3_block3_1_conv[0][0]']
 rmalization)

 conv3_block3_1_relu (Activ  (None, 28, 28, 128)          0
['conv3_block3_1_bn[0][0]']
 ation)
 conv3_block3_2_conv (Conv2  (None, 28, 28, 128)          147584
['conv3_block3_1_relu[0][0]']
 D)

 conv3_block3_2_bn (BatchNo  (None, 28, 28, 128)          512
['conv3_block3_2_conv[0][0]']
 rmalization)

 conv3_block3_2_relu (Activ  (None, 28, 28, 128)          0
['conv3_block3_2_bn[0][0]']
 ation)

 conv3_block3_3_conv (Conv2  (None, 28, 28, 512)          66048
['conv3_block3_2_relu[0][0]']
 D)

 conv3_block3_3_bn (BatchNo  (None, 28, 28, 512)          2048
['conv3_block3_3_conv[0][0]']
 rmalization)

 conv3_block3_add (Add)      (None, 28, 28, 512)          0
['conv3_block2_out[0][0]',
'conv3_block3_3_bn[0][0]']

 conv3_block3_out (Activati  (None, 28, 28, 512)          0
['conv3_block3_add[0][0]']
 on)

 conv3_block4_1_conv (Conv2  (None, 28, 28, 128)          65664
['conv3_block3_out[0][0]']
 D)

 conv3_block4_1_bn (BatchNo  (None, 28, 28, 128)          512
['conv3_block4_1_conv[0][0]']
 rmalization)

 conv3_block4_1_relu (Activ  (None, 28, 28, 128)          0
['conv3_block4_1_bn[0][0]']
 ation)

 conv3_block4_2_conv (Conv2  (None, 28, 28, 128)          147584
['conv3_block4_1_relu[0][0]']
 D)

 conv3_block4_2_bn (BatchNo  (None, 28, 28, 128)          512
['conv3_block4_2_conv[0][0]']
 rmalization)
 conv3_block4_2_relu (Activ  (None, 28, 28, 128)          0
['conv3_block4_2_bn[0][0]']
 ation)

 conv3_block4_3_conv (Conv2  (None, 28, 28, 512)          66048
['conv3_block4_2_relu[0][0]']
 D)

 conv3_block4_3_bn (BatchNo  (None, 28, 28, 512)          2048
['conv3_block4_3_conv[0][0]']
 rmalization)

 conv3_block4_add (Add)      (None, 28, 28, 512)          0
['conv3_block3_out[0][0]',
'conv3_block4_3_bn[0][0]']

 conv3_block4_out (Activati  (None, 28, 28, 512)          0
['conv3_block4_add[0][0]']
 on)

 conv4_block1_1_conv (Conv2  (None, 14, 14, 256)          131328
['conv3_block4_out[0][0]']
 D)

 conv4_block1_1_bn (BatchNo  (None, 14, 14, 256)          1024
['conv4_block1_1_conv[0][0]']
 rmalization)

 conv4_block1_1_relu (Activ  (None, 14, 14, 256)          0
['conv4_block1_1_bn[0][0]']
 ation)

 conv4_block1_2_conv (Conv2  (None, 14, 14, 256)          590080
['conv4_block1_1_relu[0][0]']
 D)

 conv4_block1_2_bn (BatchNo  (None, 14, 14, 256)          1024
['conv4_block1_2_conv[0][0]']
 rmalization)

 conv4_block1_2_relu (Activ  (None, 14, 14, 256)          0
['conv4_block1_2_bn[0][0]']
 ation)

 conv4_block1_0_conv (Conv2  (None, 14, 14, 1024)         525312
['conv3_block4_out[0][0]']
 D)
 conv4_block1_3_conv (Conv2  (None, 14, 14, 1024)         263168
['conv4_block1_2_relu[0][0]']
 D)

 conv4_block1_0_bn (BatchNo  (None, 14, 14, 1024)         4096
['conv4_block1_0_conv[0][0]']
 rmalization)

 conv4_block1_3_bn (BatchNo  (None, 14, 14, 1024)         4096
['conv4_block1_3_conv[0][0]']
 rmalization)

 conv4_block1_add (Add)      (None, 14, 14, 1024)         0
['conv4_block1_0_bn[0][0]',
'conv4_block1_3_bn[0][0]']

 conv4_block1_out (Activati  (None, 14, 14, 1024)         0
['conv4_block1_add[0][0]']
 on)

 conv4_block2_1_conv (Conv2  (None, 14, 14, 256)          262400
['conv4_block1_out[0][0]']
 D)

 conv4_block2_1_bn (BatchNo  (None, 14, 14, 256)          1024
['conv4_block2_1_conv[0][0]']
 rmalization)

 conv4_block2_1_relu (Activ  (None, 14, 14, 256)          0
['conv4_block2_1_bn[0][0]']
 ation)

 conv4_block2_2_conv (Conv2  (None, 14, 14, 256)          590080
['conv4_block2_1_relu[0][0]']
 D)

 conv4_block2_2_bn (BatchNo  (None, 14, 14, 256)          1024
['conv4_block2_2_conv[0][0]']
 rmalization)

 conv4_block2_2_relu (Activ  (None, 14, 14, 256)          0
['conv4_block2_2_bn[0][0]']
 ation)

 conv4_block2_3_conv (Conv2  (None, 14, 14, 1024)         263168
['conv4_block2_2_relu[0][0]']
 D)
 conv4_block2_3_bn (BatchNo  (None, 14, 14, 1024)         4096
['conv4_block2_3_conv[0][0]']
 rmalization)

 conv4_block2_add (Add)      (None, 14, 14, 1024)         0
['conv4_block1_out[0][0]',
'conv4_block2_3_bn[0][0]']

 conv4_block2_out (Activati  (None, 14, 14, 1024)         0
['conv4_block2_add[0][0]']
 on)

 conv4_block3_1_conv (Conv2  (None, 14, 14, 256)          262400
['conv4_block2_out[0][0]']
 D)

 conv4_block3_1_bn (BatchNo  (None, 14, 14, 256)          1024
['conv4_block3_1_conv[0][0]']
 rmalization)

 conv4_block3_1_relu (Activ  (None, 14, 14, 256)          0
['conv4_block3_1_bn[0][0]']
 ation)

 conv4_block3_2_conv (Conv2  (None, 14, 14, 256)          590080
['conv4_block3_1_relu[0][0]']
 D)

 conv4_block3_2_bn (BatchNo  (None, 14, 14, 256)          1024
['conv4_block3_2_conv[0][0]']
 rmalization)

 conv4_block3_2_relu (Activ  (None, 14, 14, 256)          0
['conv4_block3_2_bn[0][0]']
 ation)

 conv4_block3_3_conv (Conv2  (None, 14, 14, 1024)         263168
['conv4_block3_2_relu[0][0]']
 D)

 conv4_block3_3_bn (BatchNo  (None, 14, 14, 1024)         4096
['conv4_block3_3_conv[0][0]']
 rmalization)

 conv4_block3_add (Add)      (None, 14, 14, 1024)         0
['conv4_block2_out[0][0]',
'conv4_block3_3_bn[0][0]']
```

```
 conv4_block3_out (Activati  (None, 14, 14, 1024)         0
['conv4_block3_add[0][0]']
 on)

 conv4_block4_1_conv (Conv2  (None, 14, 14, 256)          262400
['conv4_block3_out[0][0]']
 D)

 conv4_block4_1_bn (BatchNo  (None, 14, 14, 256)          1024
['conv4_block4_1_conv[0][0]']
 rmalization)

 conv4_block4_1_relu (Activ  (None, 14, 14, 256)          0
['conv4_block4_1_bn[0][0]']
 ation)

 conv4_block4_2_conv (Conv2  (None, 14, 14, 256)          590080
['conv4_block4_1_relu[0][0]']
 D)

 conv4_block4_2_bn (BatchNo  (None, 14, 14, 256)          1024
['conv4_block4_2_conv[0][0]']
 rmalization)

 conv4_block4_2_relu (Activ  (None, 14, 14, 256)          0
['conv4_block4_2_bn[0][0]']
 ation)

 conv4_block4_3_conv (Conv2  (None, 14, 14, 1024)         263168
['conv4_block4_2_relu[0][0]']
 D)

 conv4_block4_3_bn (BatchNo  (None, 14, 14, 1024)         4096
['conv4_block4_3_conv[0][0]']
 rmalization)

 conv4_block4_add (Add)      (None, 14, 14, 1024)         0
['conv4_block3_out[0][0]',
'conv4_block4_3_bn[0][0]']

 conv4_block4_out (Activati  (None, 14, 14, 1024)         0
['conv4_block4_add[0][0]']
 on)

 conv4_block5_1_conv (Conv2  (None, 14, 14, 256)          262400
['conv4_block4_out[0][0]']
 D)
 conv4_block5_1_bn (BatchNo  (None, 14, 14, 256)          1024
['conv4_block5_1_conv[0][0]']
 rmalization)

 conv4_block5_1_relu (Activ  (None, 14, 14, 256)          0
['conv4_block5_1_bn[0][0]']
 ation)

 conv4_block5_2_conv (Conv2  (None, 14, 14, 256)          590080
['conv4_block5_1_relu[0][0]']
 D)

 conv4_block5_2_bn (BatchNo  (None, 14, 14, 256)          1024
['conv4_block5_2_conv[0][0]']
 rmalization)

 conv4_block5_2_relu (Activ  (None, 14, 14, 256)          0
['conv4_block5_2_bn[0][0]']
 ation)

 conv4_block5_3_conv (Conv2  (None, 14, 14, 1024)         263168
['conv4_block5_2_relu[0][0]']
 D)

 conv4_block5_3_bn (BatchNo  (None, 14, 14, 1024)         4096
['conv4_block5_3_conv[0][0]']
 rmalization)

 conv4_block5_add (Add)      (None, 14, 14, 1024)         0
['conv4_block4_out[0][0]',
'conv4_block5_3_bn[0][0]']

 conv4_block5_out (Activati  (None, 14, 14, 1024)         0
['conv4_block5_add[0][0]']
 on)

 conv4_block6_1_conv (Conv2  (None, 14, 14, 256)          262400
['conv4_block5_out[0][0]']
 D)

 conv4_block6_1_bn (BatchNo  (None, 14, 14, 256)          1024
['conv4_block6_1_conv[0][0]']
 rmalization)

 conv4_block6_1_relu (Activ  (None, 14, 14, 256)          0
['conv4_block6_1_bn[0][0]']
 ation)
 conv4_block6_2_conv (Conv2  (None, 14, 14, 256)          590080
['conv4_block6_1_relu[0][0]']
 D)

 conv4_block6_2_bn (BatchNo  (None, 14, 14, 256)          1024
['conv4_block6_2_conv[0][0]']
 rmalization)

 conv4_block6_2_relu (Activ  (None, 14, 14, 256)          0
['conv4_block6_2_bn[0][0]']
 ation)

 conv4_block6_3_conv (Conv2  (None, 14, 14, 1024)         263168
['conv4_block6_2_relu[0][0]']
 D)

 conv4_block6_3_bn (BatchNo  (None, 14, 14, 1024)         4096
['conv4_block6_3_conv[0][0]']
 rmalization)

 conv4_block6_add (Add)      (None, 14, 14, 1024)         0
['conv4_block5_out[0][0]',
'conv4_block6_3_bn[0][0]']

 conv4_block6_out (Activati  (None, 14, 14, 1024)         0
['conv4_block6_add[0][0]']
 on)

 conv5_block1_1_conv (Conv2  (None, 7, 7, 512)            524800
['conv4_block6_out[0][0]']
 D)

 conv5_block1_1_bn (BatchNo  (None, 7, 7, 512)            2048
['conv5_block1_1_conv[0][0]']
 rmalization)

 conv5_block1_1_relu (Activ  (None, 7, 7, 512)            0
['conv5_block1_1_bn[0][0]']
 ation)

 conv5_block1_2_conv (Conv2  (None, 7, 7, 512)            2359808
['conv5_block1_1_relu[0][0]']
 D)

 conv5_block1_2_bn (BatchNo  (None, 7, 7, 512)            2048
['conv5_block1_2_conv[0][0]']
 rmalization)
 conv5_block1_2_relu (Activ  (None, 7, 7, 512)            0
['conv5_block1_2_bn[0][0]']
 ation)

 conv5_block1_0_conv (Conv2  (None, 7, 7, 2048)           2099200
['conv4_block6_out[0][0]']
 D)

 conv5_block1_3_conv (Conv2  (None, 7, 7, 2048)           1050624
['conv5_block1_2_relu[0][0]']
 D)

 conv5_block1_0_bn (BatchNo  (None, 7, 7, 2048)           8192
['conv5_block1_0_conv[0][0]']
 rmalization)

 conv5_block1_3_bn (BatchNo  (None, 7, 7, 2048)           8192
['conv5_block1_3_conv[0][0]']
 rmalization)

 conv5_block1_add (Add)      (None, 7, 7, 2048)           0
['conv5_block1_0_bn[0][0]',
'conv5_block1_3_bn[0][0]']

 conv5_block1_out (Activati  (None, 7, 7, 2048)           0
['conv5_block1_add[0][0]']
 on)

 conv5_block2_1_conv (Conv2  (None, 7, 7, 512)            1049088
['conv5_block1_out[0][0]']
 D)

 conv5_block2_1_bn (BatchNo  (None, 7, 7, 512)            2048
['conv5_block2_1_conv[0][0]']
 rmalization)

 conv5_block2_1_relu (Activ  (None, 7, 7, 512)            0
['conv5_block2_1_bn[0][0]']
 ation)

 conv5_block2_2_conv (Conv2  (None, 7, 7, 512)            2359808
['conv5_block2_1_relu[0][0]']
 D)

 conv5_block2_2_bn (BatchNo  (None, 7, 7, 512)            2048
['conv5_block2_2_conv[0][0]']
 rmalization)
 conv5_block2_2_relu (Activ  (None, 7, 7, 512)            0
['conv5_block2_2_bn[0][0]']
 ation)

 conv5_block2_3_conv (Conv2  (None, 7, 7, 2048)           1050624
['conv5_block2_2_relu[0][0]']
 D)

 conv5_block2_3_bn (BatchNo  (None, 7, 7, 2048)           8192
['conv5_block2_3_conv[0][0]']
 rmalization)

 conv5_block2_add (Add)      (None, 7, 7, 2048)           0
['conv5_block1_out[0][0]',
'conv5_block2_3_bn[0][0]']

 conv5_block2_out (Activati  (None, 7, 7, 2048)           0
['conv5_block2_add[0][0]']
 on)

 conv5_block3_1_conv (Conv2  (None, 7, 7, 512)            1049088
['conv5_block2_out[0][0]']
 D)

 conv5_block3_1_bn (BatchNo  (None, 7, 7, 512)            2048
['conv5_block3_1_conv[0][0]']
 rmalization)

 conv5_block3_1_relu (Activ  (None, 7, 7, 512)            0
['conv5_block3_1_bn[0][0]']
 ation)

 conv5_block3_2_conv (Conv2  (None, 7, 7, 512)            2359808
['conv5_block3_1_relu[0][0]']
 D)

 conv5_block3_2_bn (BatchNo  (None, 7, 7, 512)            2048
['conv5_block3_2_conv[0][0]']
 rmalization)

 conv5_block3_2_relu (Activ  (None, 7, 7, 512)            0
['conv5_block3_2_bn[0][0]']
 ation)

 conv5_block3_3_conv (Conv2  (None, 7, 7, 2048)           1050624
['conv5_block3_2_relu[0][0]']
 D)
 conv5_block3_3_bn (BatchNo  (None, 7, 7, 2048)           8192
['conv5_block3_3_conv[0][0]']
 rmalization)

 conv5_block3_add (Add)      (None, 7, 7, 2048)           0
['conv5_block2_out[0][0]',
'conv5_block3_3_bn[0][0]']

 conv5_block3_out (Activati  (None, 7, 7, 2048)           0
['conv5_block3_add[0][0]']
 on)

 avg_pool (GlobalAveragePoo  (None, 2048)                 0
['conv5_block3_out[0][0]']
 ling2D)

 predictions (Dense)         (None, 1000)                 2049000
['avg_pool[0][0]']

================================================================================
==================
Total params: 25636712 (97.80 MB)
Trainable params: 25583592 (97.59 MB)
Non-trainable params: 53120 (207.50 KB)
________________________________________________________________________________
__________________
None
1/1 [==============================] - 1s 533ms/step
dog.png
[[('n02107312', 'miniature_pinscher', 0.69529486), ('n02085620', 'Chihuahua',
0.20484017), ('n02087046', 'toy_terrier', 0.07397285), ('n02105412', 'kelpie',
0.0033932144), ('n02091467', 'Norwegian_elkhound', 0.0024811204)]]
1/1 [==============================] - 0s 27ms/step
cat.png
[[('n02124075', 'Egyptian_cat', 0.43266335), ('n02123045', 'tabby', 0.26458097),
('n02123394', 'Persian_cat', 0.14259359), ('n02127052', 'lynx', 0.120225586),
('n02123159', 'tiger_cat', 0.018908592)]]
1/1 [==============================] - 0s 24ms/step
jet.png
[[('n03773504', 'missile', 0.38750985), ('n04008634', 'projectile', 0.31083208),
('n04552348', 'warplane', 0.21814711), ('n02690373', 'airliner', 0.039310187),
('n02687172', 'aircraft_carrier', 0.02145723)]]
1/1 [==============================] - 0s 20ms/step
elpresidente.png
[[('n04350905', 'suit', 0.50268537), ('n10148035', 'groom', 0.27388975),
('n04591157', 'Windsor_tie', 0.08131939), ('n02804610', 'bassoon', 0.03252296),
('n03759954', 'microphone', 0.02850474)]]
```

```
1/1 [==============================] - 0s 21ms/step
empirestate.png
[[('n02825657', 'bell_cote', 0.46291697), ('n03028079', 'church', 0.21498136),
('n02692877', 'airship', 0.1439926), ('n03877845', 'palace', 0.04187971),
('n03933933', 'pier', 0.018473094)]]
```

The predictions are roughly correct. The exact race of the animals is nnot completely correct. The jet has higher certainty for missile or projectile than for a warplane. The classification of the picture of Ulf Kristerssonas suit or groom is somehow reasonable. The same applies for the classification of the empirestate building as a church related building. The certainty ranges from 0.38 for the missile aka jet to 0.70 for the dog.

## 9 Part 8 (OPTIONAL)

Set up `Ray Tune` and run automatic hyper parameter optimization for the CNN model as we have done in the DNN lab. Remember that you have to define the `train_CNN` function, specify the hyper parameter search space and the number of samples to evaluate, among other.