pm1-convolutional-neural-networks

February 15, 2022

1 Convolutional Neural Networks

In this notebook we are going to explore the CIFAR-10 dataset (you don't need to download this dataset, we are going to use keras to download this dataset). This is a great dataset to train models for visual recognition and to start to build some models in Convolutional Neural Networks (CNN). This dataset consists of 60,000 32x32 colour images in 10 classes, with 6,000 images per class. There are 50,000 training images and 10,000 test images

As CNN's requires high-computational effort, we are going to use a reduced version of this training dataset. Given our time and computational resources restrictions, we are going to select 3 categories (airplane, horse and truck).

In this notebook, we are going to build two different models in order to classify the objects. First, we are going to build Shallow Neural Network based just in a few Fully-Connected Layers (aka Multi-layer Perceptron) and we are going to understand why is not feasible to classify images with such networks. Then, we are going to build a CNN network to perform the same task and evaluate its performance.

Again, in order to have a clean notebook, some functions are implemented in the file *utils.py* (e.g., plot_loss_and_accuracy).

Summary: - Downloading CIFAR-10 Dataset - Data Pre-processing - Reducing the Dataset - Normalising the Dataset - One-hot Encoding - Building the Shallow Neural Network - Training the Model - Prediction and Performance Analysis - Building the Convolutional Neural Network - Training the Model - Prediction and Performance Analysis

[1]:
```
# Standard libraries
import numpy as np  # written in C, is faster and robust library for numerical␣
 ↪and matrix operations
import pandas as pd # data manipulation library, it is widely used for data␣
 ↪analysis and relies on numpy library.
import matplotlib.pyplot as plt # for plotting
import seaborn as sns # Plot nicely =) . Importing seaborn modifies the default␣
 ↪matplotlib color schemes and plot
                       # styles to improve readability and aesthetics.

# Auxiliar functions
from utils import *
```

```
# the following to lines will tell to the python kernel to always update the␣
↪kernel for every utils.py
# modification, without the need of restarting the kernel.
%load_ext autoreload
%autoreload 2

# using the 'inline' backend, your matplotlib graphs will be included in your␣
↪notebook, next to the code
%matplotlib inline

import warnings
warnings.filterwarnings('ignore')
```

1.1 Downloading CIFAR-10 Dataset

Keras provides several datasets for experimentation, this makes it easy to try new network architectures. In order to download the CIFAR-10 dataset, we need to import the library "cifar10" and call the method *load_data()".

[2]:
```
from keras.datasets import cifar10 # Implements the methods to dowload CIFAR-10␣
↪dataset

(x_train, y_train), (x_test, y_test) = cifar10.load_data() #this will download␣
↪the dataset
# by defaul, the dataset was split in 50,000 images for training and 10,000␣
↪images for testing
# we are going to use this configuration

y_train = y_train.ravel() # Return a contiguous flattened y_train
y_test = y_test.ravel() #Return a contiguous flattened y_test
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170500096/170498071 [==============================] - 16s 0us/step
```

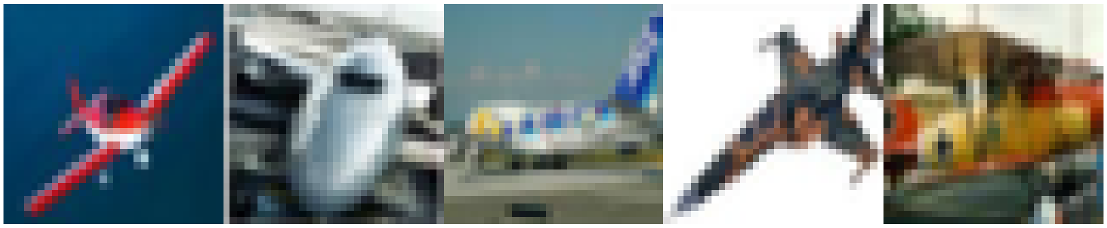
Let's visualise how the images looks like. To plot the images we are going to use the function **plot_images** (see *utils.py*)

[3]:
```
# from https://www.cs.toronto.edu/~kriz/cifar.html we can grab the class names
#                 0            1         2       3     4       5      6       7␣
↪       8       9
class_name = np.array(
    ['airplane', 'automobile','bird','cat', 'deer', 'dog', 'frog', 'horse',␣
↪'ship', 'truck'])

#
plot_samples(x_train, y_train, class_name)
```

```
---------------------------------------------------------------------------
```

----

airplane - number of samples: 5000

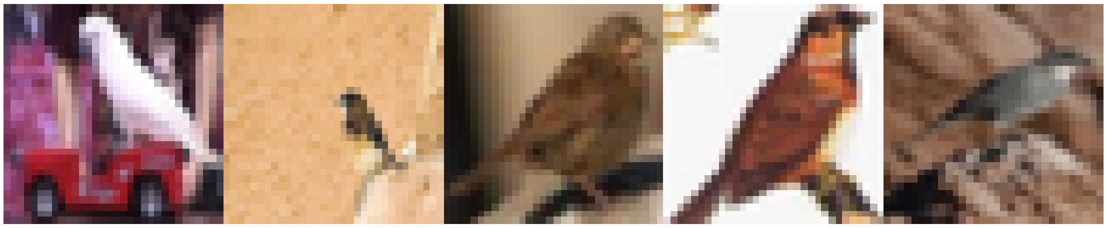


----

automobile - number of samples: 5000

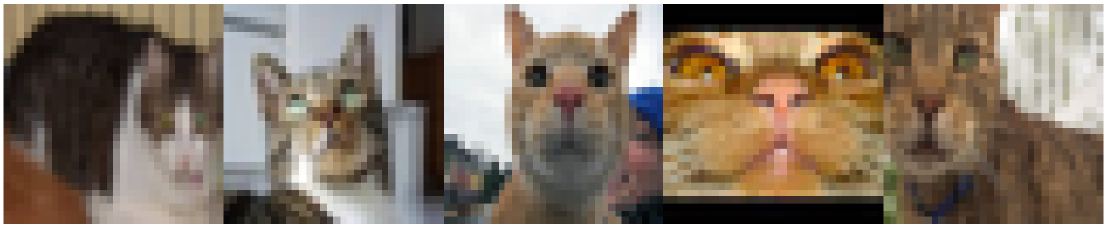
----

bird - number of samples: 5000



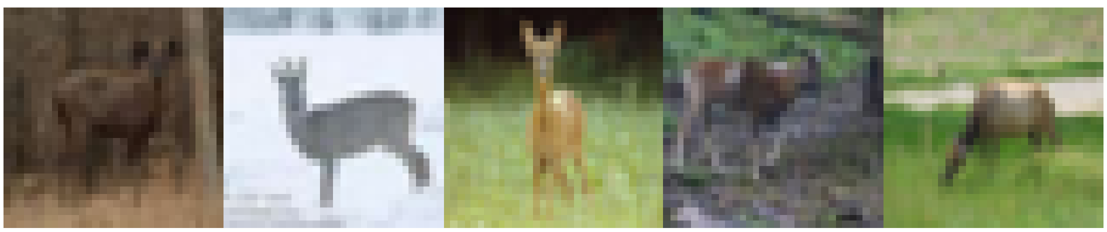
----

cat - number of samples: 5000



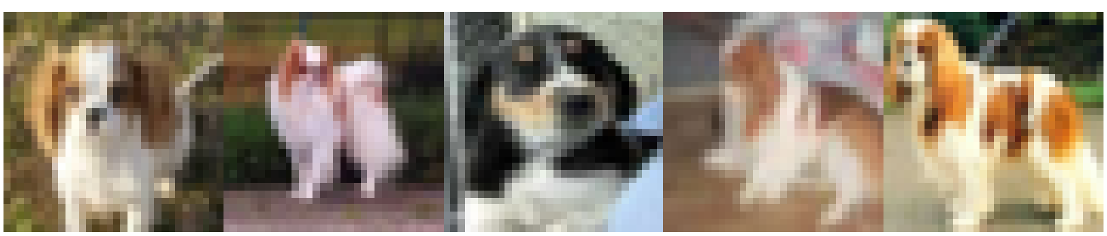
----

deer - number of samples: 5000



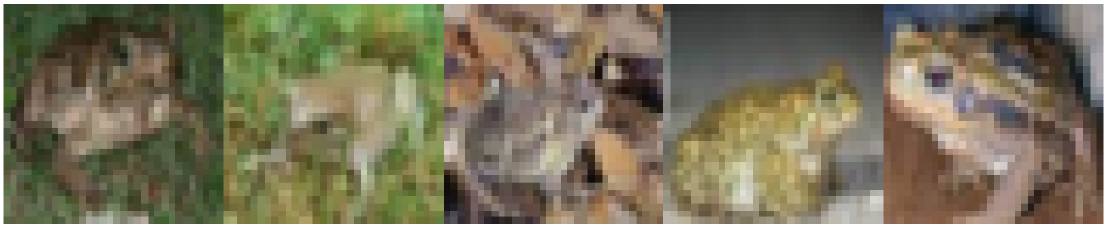
----

dog - number of samples: 5000



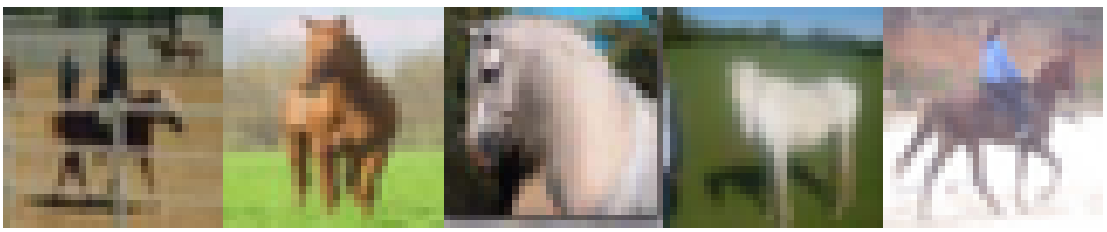
----

frog - number of samples: 5000



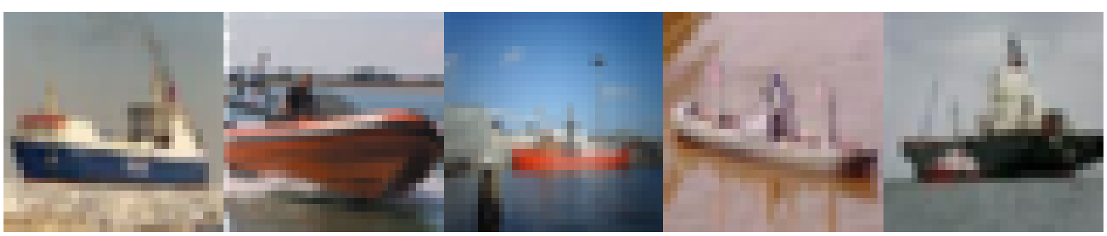
----

horse - number of samples: 5000



----

ship - number of samples: 5000



----

truck - number of samples: 5000

1.2 Data Pre-processing

As CNN's requires high-computational effort, we are going to use a reduced training dataset. Given our time and computational resources restrictions, we are going to select 3 categories (airplane, horse and truck) and for each category and select in total 1500 images.

Once obtained the reduced version, we are going to normalise the images and generate the one-hot enconding representation of the labels.

1.2.1 Reducing the Dataset

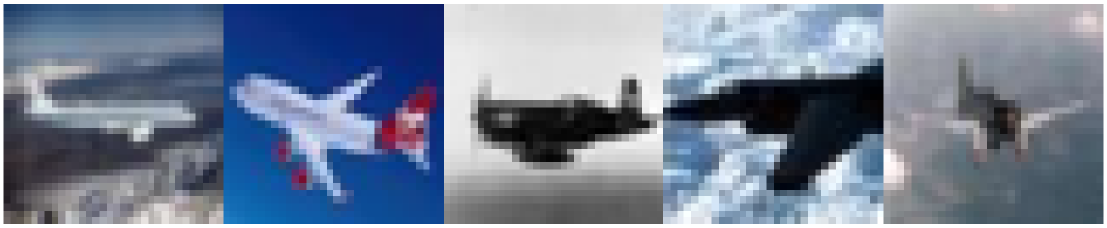
[4]:
```
# Lets select just 3 classes to make this tutorial feasible
selected_idx = np.array([0, 7, 9])
n_images = 1500

y_train_idx = np.isin(y_train, selected_idx)
y_test_idx = np.isin(y_test, selected_idx)

y_train_red = y_train[y_train_idx][:n_images]
x_train_red = x_train[y_train_idx][:n_images]

y_test_red = y_test[y_test_idx][:n_images]
x_test_red = x_test[y_test_idx][:n_images]

# replacing the labels 0, 7 and 9 to 0, 1, 2 repectively.
y_train_red[y_train_red == selected_idx[0]] = 0
y_train_red[y_train_red == selected_idx[1]] = 1
y_train_red[y_train_red == selected_idx[2]] = 2

y_test_red[y_test_red == selected_idx[0]] = 0
y_test_red[y_test_red == selected_idx[1]] = 1
y_test_red[y_test_red == selected_idx[2]] = 2
```

[5]:
```
y_test_red[:4]
```

[5]:
```
array([0, 0, 2, 1], dtype=uint8)
```

[6]:
```
# visulising the images in the reduced dataset
plot_samples(x_train_red, y_train_red, class_name[selected_idx])
```

```
--------------------------------------------------------------------------------
------------

airplane - number of samples: 508
```



```
--------------------------------------------------------------------------------
------------

horse - number of samples: 489
```

```
--------------------------------------------------------------------------------
------------

truck - number of samples: 503
```

Question 1: Is the reduced dataset imbalanced?

Question 2: As you can see, the images have low resolution (32x32x3), how this can affect the model?

1.2.2 Normalising the Dataset

Here we are going to normalise the dataset. In this task, we are going to divide each image by 255.0, as the images are represented as 'uint8' and we know that the range is from 0 to 255. By doing so, the range of the images will be between 0 and 1.

[7]:
```
# Normalising the
x_train_red = x_train_red.astype('float32')
x_test_red = x_test_red.astype('float32')
x_train_red /= 255.0
x_test_red /= 255.0
```

1.2.3 One-hot Encoding

The labels are encoded as integers (0, 1 and 2), as we are going to use a *softmax layer* as output for our models we need to convert the labels as binary matrix. For example, the label 0 (considering that we have just 3 classes) can be represented as [1 0 0], which is the class 0.

One-hot enconding together with the sofmax function will give us an interesting interpretation of the output as a probability distribution over the classes.

For this task, are going to use the function *to_categorical*, which converts a class vector (integers) to binary class matrix.

[8]:
```
y_train_oh = keras.utils.to_categorical(y_train_red)
y_test_oh = keras.utils.to_categorical(y_test_red)

print('Label: ',y_train_red[0], ' one-hot: ', y_train_oh[0])
print('Label: ',y_train_red[810], ' one-hot: ',  y_train_oh[810])
print('Label: ',y_test_red[20], ' one-hot: ', y_test_oh[20])
```

```
Label:  2  one-hot:  [0. 0. 1.]
Label:  0  one-hot:  [1. 0. 0.]
Label:  1  one-hot:  [0. 1. 0.]
```

1.3 Building the Shallow Neural Network

Here we are going to build a Shallow Neural Network with 2 Fully Connected layers and one output layer. Basically, we are implemting a Multi-Layer Perceptron classifier.

To build the model, we are going use the following components from Keras:

- Sequencial: allows us to create models layer-by-layer.
- Dense: provides a regular fully-connected layer
- Dropout: provides dropout regularisation

Basically, we are going to define the sequence of our model by using *Sequential()*, which include the layers:

```
model = Sequential()
model.add(Dense(...))
...
```

once created the model we can configure the model for training by using the method compile. Here we need to define the loss function (mean squared error, categorical cross entropy, among others.), the optimizer (Stochastic gradient descent, RMSprop, adam, among others) and the metric to define the evaluation metric to be used to evaluate the performance of the model in the training step, as follows:

```
model.compile(loss = "...",
              optimizer = "...")
```

Also, we have the option to see a summary representation of the model by using thebfunction summary. This function summarise the model and tell us the number of parameters that we need to tune.

[9]:
```
from keras.models import Sequential # implements sequential function
from keras.layers import Dense # implements the fully connected layer
from keras.layers import Dropout # implements Dropout regularisation
from keras.layers import Flatten # implements Flatten function
```

[10]:
```
mlp = Sequential()

# Flatten will reshape the input an 1D array with dimension equal to 32 x 32 x␣
↪3 (3072)
# each pixel is an input for this model.
mlp.add(Flatten(input_shape=x_train_red.shape[1:])) #x_train.shape[1:] returns␣
↪the shape

# First layer with 1024 neurons and relu as activation function
mlp.add(Dense(1024, activation='relu'))
mlp.add(Dropout(0.7)) # regularization with 70% of keep probability

# Second layer with 1024 neurons and relu as activation function
mlp.add(Dense(1024, activation='relu'))
mlp.add(Dropout(0.7))# regularization with 70% of keep probability

# Output layer with 3 neurons and sofmax as activation function
mlp.add(Dense(y_test_oh.shape[1], activation='softmax'))
```

```
2022-02-03 06:26:01.053415: I tensorflow/compiler/jit/xla_cpu_device.cc:41] Not
creating XLA devices, tf_xla_enable_xla_devices not set
2022-02-03 06:26:01.053611: I tensorflow/core/platform/cpu_feature_guard.cc:142]
This TensorFlow binary is optimized with oneAPI Deep Neural Network Library
(oneDNN) to use the following CPU instructions in performance-critical
operations:  SSE4.1 SSE4.2 AVX AVX2 AVX512F FMA
To enable them in other operations, rebuild TensorFlow with the appropriate
compiler flags.
```

```
2022-02-03 06:26:01.055599: I
tensorflow/core/common_runtime/process_util.cc:146] Creating new thread pool
with default inter op setting: 2. Tune using inter_op_parallelism_threads for
best performance.
```

Summarising the model

[11]:
```
mlp.summary()
```

```
Model: "sequential"

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 3072)              0

_________________________________________________________________
dense (Dense)                (None, 1024)              3146752

_________________________________________________________________
dropout (Dropout)            (None, 1024)              0

_________________________________________________________________
dense_1 (Dense)              (None, 1024)              1049600

_________________________________________________________________
dropout_1 (Dropout)          (None, 1024)              0

_________________________________________________________________
dense_2 (Dense)              (None, 3)                 3075
=================================================================
Total params: 4,199,427
Trainable params: 4,199,427
Non-trainable params: 0

_________________________________________________________________
```

[12]:
```
# Compile:
#  Optimiser: rmsprop
#  Loss: categorical_crossentropy, as our problem is multi-label classification
#  Metric: accuracy

mlp.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

1.3.1 Training the Model

Once defined the model, we need to train it by using the function fit. This function performs the optmisation step. Hence, we can define the following parameters such as:

- batch size: defines the number of samples that will be propagated through the network
- epochs: defines the number of times in which all the training set (x_train_scaled) are used once to update the weights
- validation split: defines the percentage of training data to be used for validation
- among others (click here for more information)

This function return the *history* of the training, that can be used for further performance analysis.

[13]:
```
# training the model (this will take a few minutes)
history = mlp.fit(x_train_red,
                    y_train_oh,
                    batch_size = 256,
                    epochs = 100,
                    validation_split = 0.2,
                    verbose = 1)
```

```
Epoch 1/100
```

```
2022-02-03 06:26:01.286745: I
tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:116] None of the MLIR
optimization passes are enabled (registered 2)
2022-02-03 06:26:01.306311: I
tensorflow/core/platform/profile_utils/cpu_utils.cc:112] CPU Frequency:
2599990000 Hz
```

```
5/5 [==============================] - 2s 242ms/step - loss: 12.7446 - accuracy:
0.3246 - val_loss: 3.3273 - val_accuracy: 0.2933
Epoch 2/100
5/5 [==============================] - 0s 68ms/step - loss: 2.7455 - accuracy:
0.3708 - val_loss: 1.0823 - val_accuracy: 0.4900
Epoch 3/100
5/5 [==============================] - 0s 90ms/step - loss: 1.3511 - accuracy:
0.4141 - val_loss: 1.0248 - val_accuracy: 0.5100
Epoch 4/100
5/5 [==============================] - 0s 106ms/step - loss: 1.0795 - accuracy:
0.4324 - val_loss: 0.9979 - val_accuracy: 0.5467
Epoch 5/100
5/5 [==============================] - 0s 110ms/step - loss: 1.1250 - accuracy:
0.3966 - val_loss: 1.0281 - val_accuracy: 0.4567
Epoch 6/100
5/5 [==============================] - 0s 91ms/step - loss: 1.0867 - accuracy:
0.4691 - val_loss: 1.0896 - val_accuracy: 0.4167
Epoch 7/100
5/5 [==============================] - 1s 102ms/step - loss: 1.1937 - accuracy:
0.4375 - val_loss: 1.0896 - val_accuracy: 0.4467
Epoch 8/100
5/5 [==============================] - 0s 96ms/step - loss: 1.0910 - accuracy:
0.4573 - val_loss: 0.9952 - val_accuracy: 0.5300
Epoch 9/100
5/5 [==============================] - 1s 124ms/step - loss: 1.1044 - accuracy:
0.4766 - val_loss: 1.0239 - val_accuracy: 0.4033
Epoch 10/100
5/5 [==============================] - 0s 98ms/step - loss: 1.0839 - accuracy:
0.4562 - val_loss: 0.9577 - val_accuracy: 0.5633
Epoch 11/100
```

```
5/5 [==============================] - 0s 96ms/step - loss: 1.0780 - accuracy:
0.4843 - val_loss: 1.0976 - val_accuracy: 0.4233
Epoch 12/100
5/5 [==============================] - 1s 110ms/step - loss: 1.1463 - accuracy:
0.4566 - val_loss: 0.9380 - val_accuracy: 0.5367
Epoch 13/100
5/5 [==============================] - 0s 99ms/step - loss: 1.0395 - accuracy:
0.4931 - val_loss: 0.9331 - val_accuracy: 0.5467
Epoch 14/100
5/5 [==============================] - 1s 146ms/step - loss: 1.1593 - accuracy:
0.4457 - val_loss: 1.0575 - val_accuracy: 0.4233
Epoch 15/100
5/5 [==============================] - 1s 104ms/step - loss: 1.0212 - accuracy:
0.4947 - val_loss: 0.8938 - val_accuracy: 0.6233
Epoch 16/100
5/5 [==============================] - 1s 127ms/step - loss: 1.0271 - accuracy:
0.5437 - val_loss: 1.1380 - val_accuracy: 0.4133
Epoch 17/100
5/5 [==============================] - 0s 104ms/step - loss: 1.1130 - accuracy:
0.5055 - val_loss: 0.8787 - val_accuracy: 0.6100
Epoch 18/100
5/5 [==============================] - 1s 131ms/step - loss: 0.9650 - accuracy:
0.5566 - val_loss: 0.9733 - val_accuracy: 0.5400
Epoch 19/100
5/5 [==============================] - 1s 94ms/step - loss: 0.9369 - accuracy:
0.5586 - val_loss: 0.9285 - val_accuracy: 0.5400
Epoch 20/100
5/5 [==============================] - 1s 176ms/step - loss: 0.9232 - accuracy:
0.5543 - val_loss: 0.9510 - val_accuracy: 0.5867
Epoch 21/100
5/5 [==============================] - 0s 101ms/step - loss: 0.9982 - accuracy:
0.5318 - val_loss: 0.9994 - val_accuracy: 0.5100
Epoch 22/100
5/5 [==============================] - 1s 128ms/step - loss: 0.9497 - accuracy:
0.5487 - val_loss: 0.8976 - val_accuracy: 0.5533
Epoch 23/100
5/5 [==============================] - 1s 119ms/step - loss: 0.9527 - accuracy:
0.5578 - val_loss: 0.8174 - val_accuracy: 0.6500
Epoch 24/100
5/5 [==============================] - 1s 120ms/step - loss: 0.9728 - accuracy:
0.5620 - val_loss: 0.8937 - val_accuracy: 0.6367
Epoch 25/100
5/5 [==============================] - 0s 101ms/step - loss: 0.9284 - accuracy:
0.5964 - val_loss: 0.8774 - val_accuracy: 0.6200
Epoch 26/100
5/5 [==============================] - 0s 94ms/step - loss: 0.8647 - accuracy:
0.6140 - val_loss: 1.0172 - val_accuracy: 0.5133
Epoch 27/100
```

```
5/5 [==============================] - 0s 99ms/step - loss: 1.0143 - accuracy:
0.5578 - val_loss: 0.8986 - val_accuracy: 0.5833
Epoch 28/100
5/5 [==============================] - 1s 144ms/step - loss: 0.9208 - accuracy:
0.5726 - val_loss: 0.8733 - val_accuracy: 0.6567
Epoch 29/100
5/5 [==============================] - 0s 107ms/step - loss: 0.8506 - accuracy:
0.6077 - val_loss: 0.8426 - val_accuracy: 0.6733
Epoch 30/100
5/5 [==============================] - 0s 97ms/step - loss: 0.9402 - accuracy:
0.5653 - val_loss: 0.8869 - val_accuracy: 0.6733
Epoch 31/100
5/5 [==============================] - 0s 69ms/step - loss: 0.8952 - accuracy:
0.5855 - val_loss: 0.9901 - val_accuracy: 0.5733
Epoch 32/100
5/5 [==============================] - 1s 157ms/step - loss: 1.0194 - accuracy:
0.5800 - val_loss: 0.8584 - val_accuracy: 0.6167
Epoch 33/100
5/5 [==============================] - 1s 157ms/step - loss: 0.9459 - accuracy:
0.6013 - val_loss: 0.8881 - val_accuracy: 0.6400
Epoch 34/100
5/5 [==============================] - 0s 104ms/step - loss: 0.8842 - accuracy:
0.6157 - val_loss: 0.8478 - val_accuracy: 0.6167
Epoch 35/100
5/5 [==============================] - 0s 97ms/step - loss: 0.9221 - accuracy:
0.5927 - val_loss: 0.8655 - val_accuracy: 0.6400
Epoch 36/100
5/5 [==============================] - 0s 98ms/step - loss: 0.8877 - accuracy:
0.6199 - val_loss: 0.8082 - val_accuracy: 0.6600
Epoch 37/100
5/5 [==============================] - 0s 71ms/step - loss: 0.8423 - accuracy:
0.6457 - val_loss: 0.8204 - val_accuracy: 0.6900
Epoch 38/100
5/5 [==============================] - 0s 86ms/step - loss: 0.8213 - accuracy:
0.6366 - val_loss: 0.7999 - val_accuracy: 0.7033
Epoch 39/100
5/5 [==============================] - 0s 79ms/step - loss: 0.8452 - accuracy:
0.6370 - val_loss: 0.8313 - val_accuracy: 0.6467
Epoch 40/100
5/5 [==============================] - 0s 94ms/step - loss: 0.8110 - accuracy:
0.6285 - val_loss: 0.9713 - val_accuracy: 0.5733
Epoch 41/100
5/5 [==============================] - 0s 75ms/step - loss: 1.0475 - accuracy:
0.5363 - val_loss: 0.8222 - val_accuracy: 0.6900
Epoch 42/100
5/5 [==============================] - 0s 72ms/step - loss: 0.8318 - accuracy:
0.6204 - val_loss: 0.8369 - val_accuracy: 0.6533
Epoch 43/100
```

```
5/5 [==============================] - 0s 96ms/step - loss: 0.8452 - accuracy:
0.6445 - val_loss: 0.8077 - val_accuracy: 0.6533
Epoch 44/100
5/5 [==============================] - 0s 90ms/step - loss: 0.9077 - accuracy:
0.5823 - val_loss: 0.8204 - val_accuracy: 0.6767
Epoch 45/100
5/5 [==============================] - 0s 82ms/step - loss: 0.8397 - accuracy:
0.6315 - val_loss: 0.8177 - val_accuracy: 0.6900
Epoch 46/100
5/5 [==============================] - 0s 81ms/step - loss: 0.7865 - accuracy:
0.6784 - val_loss: 0.8771 - val_accuracy: 0.6167
Epoch 47/100
5/5 [==============================] - 0s 92ms/step - loss: 0.7626 - accuracy:
0.6753 - val_loss: 0.8544 - val_accuracy: 0.6100
Epoch 48/100
5/5 [==============================] - 0s 93ms/step - loss: 0.8158 - accuracy:
0.6531 - val_loss: 0.8250 - val_accuracy: 0.6967
Epoch 49/100
5/5 [==============================] - 0s 79ms/step - loss: 0.8323 - accuracy:
0.6349 - val_loss: 1.1972 - val_accuracy: 0.4400
Epoch 50/100
5/5 [==============================] - 0s 79ms/step - loss: 1.1040 - accuracy:
0.5640 - val_loss: 0.8621 - val_accuracy: 0.5900
Epoch 51/100
5/5 [==============================] - 0s 88ms/step - loss: 0.7946 - accuracy:
0.6573 - val_loss: 0.7982 - val_accuracy: 0.7167
Epoch 52/100
5/5 [==============================] - 0s 100ms/step - loss: 0.7303 - accuracy:
0.6825 - val_loss: 0.7690 - val_accuracy: 0.6733
Epoch 53/100
5/5 [==============================] - 0s 86ms/step - loss: 0.8976 - accuracy:
0.6279 - val_loss: 0.8673 - val_accuracy: 0.6067
Epoch 54/100
5/5 [==============================] - 0s 84ms/step - loss: 0.8083 - accuracy:
0.6365 - val_loss: 0.9079 - val_accuracy: 0.6033
Epoch 55/100
5/5 [==============================] - 0s 82ms/step - loss: 0.8927 - accuracy:
0.6297 - val_loss: 0.9353 - val_accuracy: 0.5767
Epoch 56/100
5/5 [==============================] - 0s 73ms/step - loss: 0.8393 - accuracy:
0.6331 - val_loss: 0.7825 - val_accuracy: 0.7067
Epoch 57/100
5/5 [==============================] - 0s 87ms/step - loss: 0.7420 - accuracy:
0.7027 - val_loss: 0.9091 - val_accuracy: 0.5700
Epoch 58/100
5/5 [==============================] - 0s 76ms/step - loss: 0.8199 - accuracy:
0.6368 - val_loss: 0.8096 - val_accuracy: 0.6567
Epoch 59/100
```

```
5/5 [==============================] - 0s 95ms/step - loss: 0.8383 - accuracy:
0.6047 - val_loss: 0.8277 - val_accuracy: 0.6767
Epoch 60/100
5/5 [==============================] - 0s 111ms/step - loss: 0.7865 - accuracy:
0.6835 - val_loss: 0.8366 - val_accuracy: 0.6667
Epoch 61/100
5/5 [==============================] - 0s 111ms/step - loss: 0.7348 - accuracy:
0.6808 - val_loss: 0.8082 - val_accuracy: 0.6300
Epoch 62/100
5/5 [==============================] - 1s 140ms/step - loss: 1.1272 - accuracy:
0.5757 - val_loss: 0.8188 - val_accuracy: 0.6833
Epoch 63/100
5/5 [==============================] - 0s 79ms/step - loss: 0.8080 - accuracy:
0.6549 - val_loss: 0.8336 - val_accuracy: 0.6533
Epoch 64/100
5/5 [==============================] - 0s 99ms/step - loss: 0.8246 - accuracy:
0.6416 - val_loss: 0.8231 - val_accuracy: 0.6933
Epoch 65/100
5/5 [==============================] - 0s 88ms/step - loss: 0.7997 - accuracy:
0.6455 - val_loss: 0.8241 - val_accuracy: 0.6933
Epoch 66/100
5/5 [==============================] - 0s 71ms/step - loss: 0.7876 - accuracy:
0.6691 - val_loss: 0.8171 - val_accuracy: 0.6833
Epoch 67/100
5/5 [==============================] - 0s 76ms/step - loss: 0.7971 - accuracy:
0.6675 - val_loss: 0.8611 - val_accuracy: 0.6033
Epoch 68/100
5/5 [==============================] - 0s 78ms/step - loss: 0.7795 - accuracy:
0.6582 - val_loss: 0.8422 - val_accuracy: 0.6367
Epoch 69/100
5/5 [==============================] - 0s 75ms/step - loss: 0.7262 - accuracy:
0.7027 - val_loss: 0.9163 - val_accuracy: 0.5933
Epoch 70/100
5/5 [==============================] - 0s 107ms/step - loss: 0.8961 - accuracy:
0.5983 - val_loss: 0.8123 - val_accuracy: 0.6967
Epoch 71/100
5/5 [==============================] - 0s 84ms/step - loss: 0.7735 - accuracy:
0.6855 - val_loss: 0.7827 - val_accuracy: 0.7100
Epoch 72/100
5/5 [==============================] - 0s 87ms/step - loss: 0.7325 - accuracy:
0.6964 - val_loss: 0.8773 - val_accuracy: 0.6133
Epoch 73/100
5/5 [==============================] - 0s 78ms/step - loss: 0.7924 - accuracy:
0.6700 - val_loss: 0.8183 - val_accuracy: 0.6800
Epoch 74/100
5/5 [==============================] - 0s 86ms/step - loss: 0.7211 - accuracy:
0.7126 - val_loss: 0.8447 - val_accuracy: 0.6200
Epoch 75/100
```

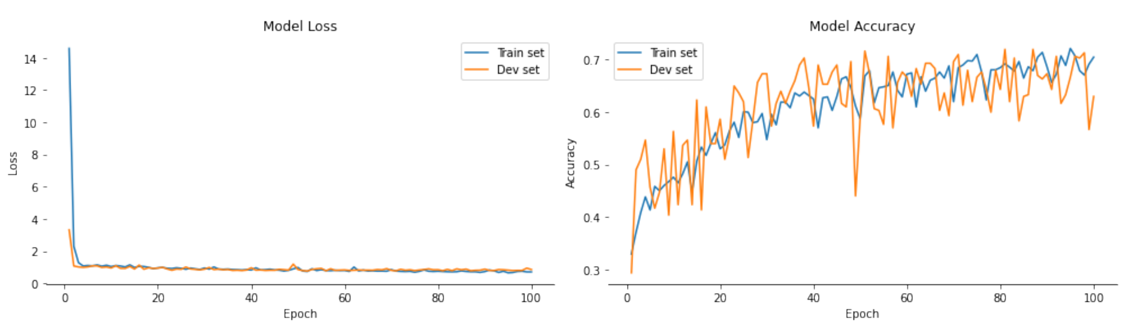
```
5/5 [==============================] - 0s 72ms/step - loss: 0.6894 - accuracy:
0.7114 - val_loss: 0.7964 - val_accuracy: 0.6667
Epoch 76/100
5/5 [==============================] - 0s 78ms/step - loss: 0.7235 - accuracy:
0.6875 - val_loss: 0.8316 - val_accuracy: 0.6767
Epoch 77/100
5/5 [==============================] - 0s 60ms/step - loss: 0.8337 - accuracy:
0.6220 - val_loss: 0.8630 - val_accuracy: 0.6433
Epoch 78/100
5/5 [==============================] - 0s 64ms/step - loss: 0.7563 - accuracy:
0.6724 - val_loss: 0.9025 - val_accuracy: 0.6000
Epoch 79/100
5/5 [==============================] - 0s 70ms/step - loss: 0.7763 - accuracy:
0.6601 - val_loss: 0.8461 - val_accuracy: 0.6800
Epoch 80/100
5/5 [==============================] - 0s 94ms/step - loss: 0.7589 - accuracy:
0.6816 - val_loss: 0.8499 - val_accuracy: 0.6433
Epoch 81/100
5/5 [==============================] - 0s 89ms/step - loss: 0.7103 - accuracy:
0.6969 - val_loss: 0.7904 - val_accuracy: 0.7200
Epoch 82/100
5/5 [==============================] - 0s 82ms/step - loss: 0.7121 - accuracy:
0.6889 - val_loss: 0.8724 - val_accuracy: 0.6200
Epoch 83/100
5/5 [==============================] - 0s 84ms/step - loss: 0.7545 - accuracy:
0.6552 - val_loss: 0.7924 - val_accuracy: 0.7033
Epoch 84/100
5/5 [==============================] - 0s 80ms/step - loss: 0.7171 - accuracy:
0.6987 - val_loss: 0.9027 - val_accuracy: 0.5833
Epoch 85/100
5/5 [==============================] - 0s 75ms/step - loss: 0.7766 - accuracy:
0.6742 - val_loss: 0.8461 - val_accuracy: 0.6300
Epoch 86/100
5/5 [==============================] - 0s 68ms/step - loss: 0.7922 - accuracy:
0.6644 - val_loss: 0.8885 - val_accuracy: 0.6333
Epoch 87/100
5/5 [==============================] - 0s 94ms/step - loss: 0.7450 - accuracy:
0.6649 - val_loss: 0.7879 - val_accuracy: 0.7200
Epoch 88/100
5/5 [==============================] - 0s 93ms/step - loss: 0.7458 - accuracy:
0.7027 - val_loss: 0.8138 - val_accuracy: 0.6700
Epoch 89/100
5/5 [==============================] - 0s 89ms/step - loss: 0.6940 - accuracy:
0.7146 - val_loss: 0.8192 - val_accuracy: 0.6633
Epoch 90/100
5/5 [==============================] - 0s 88ms/step - loss: 0.7295 - accuracy:
0.6835 - val_loss: 0.8872 - val_accuracy: 0.6733
Epoch 91/100
```

```
5/5 [==============================] - 0s 88ms/step - loss: 0.8761 - accuracy:
0.6507 - val_loss: 0.8082 - val_accuracy: 0.6433
Epoch 92/100
5/5 [==============================] - 0s 106ms/step - loss: 0.8234 - accuracy:
0.6574 - val_loss: 0.8014 - val_accuracy: 0.7067
Epoch 93/100
5/5 [==============================] - 0s 73ms/step - loss: 0.6956 - accuracy:
0.6974 - val_loss: 0.8612 - val_accuracy: 0.6167
Epoch 94/100
5/5 [==============================] - 0s 75ms/step - loss: 0.7289 - accuracy:
0.6983 - val_loss: 0.8467 - val_accuracy: 0.6333
Epoch 95/100
5/5 [==============================] - 0s 81ms/step - loss: 0.6542 - accuracy:
0.7168 - val_loss: 0.8226 - val_accuracy: 0.6667
Epoch 96/100
5/5 [==============================] - 0s 84ms/step - loss: 0.6767 - accuracy:
0.7053 - val_loss: 0.7987 - val_accuracy: 0.7067
Epoch 97/100
5/5 [==============================] - 0s 86ms/step - loss: 0.7361 - accuracy:
0.6705 - val_loss: 0.8075 - val_accuracy: 0.7033
Epoch 98/100
5/5 [==============================] - 0s 66ms/step - loss: 0.8144 - accuracy:
0.6517 - val_loss: 0.8052 - val_accuracy: 0.7133
Epoch 99/100
5/5 [==============================] - 0s 75ms/step - loss: 0.7034 - accuracy:
0.7010 - val_loss: 0.9501 - val_accuracy: 0.5667
Epoch 100/100
5/5 [==============================] - 0s 58ms/step - loss: 0.7338 - accuracy:
0.6951 - val_loss: 0.8620 - val_accuracy: 0.6300
```

1.3.2 Prediction and Performance Analysis

Here we plot the 'loss' and the 'Accuracy' from the training step.

[14]:
```
plot_loss_and_accuracy_am2(history=history)
```



Let's evaluate the performance of this model under unseen data (x_test)

[15]:
```
loss_value_mlp, acc_value_mlp = mlp.evaluate(x_test_red, y_test_oh, verbose=0)
print('Loss value: ', loss_value_mlp)
print('Acurracy value: ', acc_value_mlp)
```

```
Loss value:  0.7920286655426025
Acurracy value:  0.659333348274231
```

1.4 Building the Convolutional Neural Network

Here we are going to build a Convolutional Neural Network (CNN) for image classification. Given the time and computational resources limitations, we are going to build a very simple CNN, however, more complex and deep CNN's architectures such as VGG, Inception and ResNet are the state of the art in computer vision and they superpass the human performance in image classification tasks.

To build the model, we are going use the following components from Keras:

- Sequencial: allows us to create models layer-by-layer.
- Dense: provides a regular fully-connected layer
- Dropout: provides dropout regularisation
- Conv2D: implement 2D convolution function
- BatchNormalization: normalize the activations of the previous layer at each batch
- MaxPooling2D: provides pooling operation for spatial data

Basically, we are going to define the sequence of our model by using *Sequential()*, which include the layers:

```
model = Sequential()
model.add(Conv2D(...))
...
```

once created the model the training configuration is the same as before:

```
model.compile(loss = "...",
              optimizer = "...")
```

[16]:
```
from keras.models import Sequential
from keras.layers import Dense, Flatten, Activation
from keras.layers import Dropout, Conv2D, MaxPooling2D, BatchNormalization
```

[17]:
```
model_cnn = Sequential()

# First layer:
#  2D convolution:
#     Depth: 32
#     Kernel shape: 3 x 3
#     Stride: 1 (default)
#     Activation layer: relu
#     Padding: valid
#     Input shape: 32 x 32 x 3 (3D representation, not Flatten as MLP)
```

```
# as you can see now the input is an image and not an flattened array
model_cnn.add(Conv2D(32, (3, 3), padding='valid', activation = 'relu',
                 input_shape=x_train_red.shape[1:]))
model_cnn.add(BatchNormalization())
model_cnn.add(MaxPooling2D(pool_size=(5,5))) # max pooling with kernel size 5x5
model_cnn.add(Dropout(0.7)) # 70% of keep probability


# Second layer:
#  2D convolution:
#     Depth: 64
#     Kernel shape: 3 x 3
#     Stride: 1 (default)
#     Activation layer: relu
#     Padding: valid
model_cnn.add(Conv2D(64, (3, 3), padding='valid', activation = 'relu'))
model_cnn.add(BatchNormalization())
model_cnn.add(MaxPooling2D(pool_size=(2,2)))
model_cnn.add(Dropout(0.7))

# Flatten the output from the second layer to become the input of the␣
↪Fully-connected
# layer (flattened representation as MLP)
model_cnn.add(Flatten())

# First fully-connected layer with 128 neurons and relu as activation function
model_cnn.add(Dense(128, activation = 'relu'))

# Output layer with 3 neurons and sofmax as activation function
model_cnn.add(Dense(y_test_oh.shape[1], activation='softmax'))
```

Summarising the model

[18]:
```
model_cnn.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 30, 30, 32)        896
_________________________________________________________________
batch_normalization (BatchNo (None, 30, 30, 32)        128
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 6, 6, 32)          0
_________________________________________________________________
dropout_2 (Dropout)          (None, 6, 6, 32)          0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 4, 4, 64)          18496
```

```
_________________________________________________________________
batch_normalization_1 (Batch (None, 4, 4, 64)          256
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 2, 2, 64)          0
_________________________________________________________________
dropout_3 (Dropout)          (None, 2, 2, 64)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 256)               0
_________________________________________________________________
dense_3 (Dense)              (None, 128)               32896
_________________________________________________________________
dense_4 (Dense)              (None, 3)                 387
=================================================================
Total params: 53,059
Trainable params: 52,867
Non-trainable params: 192
_________________________________________________________________
```

As you can see, the CNN model (53,059 parameters) has less parameters than the MLP model (4,199,427 parameters). So this model is less prone to overfit.

[19]:
```
# Compile:
#  Optimiser: adam
#  Loss: categorical_crossentropy, as our problem is multi-label classification
#  Metric: accuracy

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

1.4.1 Training the Model

[20]:
```
# this will take a few minutes
history_cnn = model_cnn.fit(x_train_red,
                    y_train_oh,
                    batch_size = 256,
                    epochs = 100,
                    validation_split = 0.2,
                    verbose = 1)
```

```
Epoch 1/100
5/5 [==============================] - 3s 455ms/step - loss: 2.2466 - accuracy:
0.3548 - val_loss: 1.0973 - val_accuracy: 0.3367
Epoch 2/100
5/5 [==============================] - 2s 499ms/step - loss: 1.7110 - accuracy:
0.4097 - val_loss: 1.0887 - val_accuracy: 0.3733
Epoch 3/100
5/5 [==============================] - 3s 537ms/step - loss: 1.5097 - accuracy:
```

```
0.4602 - val_loss: 1.0824 - val_accuracy: 0.3633
Epoch 4/100
5/5 [==============================] - 3s 540ms/step - loss: 1.4123 - accuracy:
0.4973 - val_loss: 1.0757 - val_accuracy: 0.3667
Epoch 5/100
5/5 [==============================] - 3s 578ms/step - loss: 1.2367 - accuracy:
0.5290 - val_loss: 1.0689 - val_accuracy: 0.3667
Epoch 6/100
5/5 [==============================] - 3s 575ms/step - loss: 1.1683 - accuracy:
0.5753 - val_loss: 1.0651 - val_accuracy: 0.3667
Epoch 7/100
5/5 [==============================] - 2s 479ms/step - loss: 1.0694 - accuracy:
0.5790 - val_loss: 1.0604 - val_accuracy: 0.3667
Epoch 8/100
5/5 [==============================] - 2s 502ms/step - loss: 1.0551 - accuracy:
0.5867 - val_loss: 1.0542 - val_accuracy: 0.3633
Epoch 9/100
5/5 [==============================] - 3s 523ms/step - loss: 1.0268 - accuracy:
0.6022 - val_loss: 1.0472 - val_accuracy: 0.3667
Epoch 10/100
5/5 [==============================] - 2s 474ms/step - loss: 0.9536 - accuracy:
0.6061 - val_loss: 1.0459 - val_accuracy: 0.3667
Epoch 11/100
5/5 [==============================] - 3s 529ms/step - loss: 0.9457 - accuracy:
0.6310 - val_loss: 1.0447 - val_accuracy: 0.3733
Epoch 12/100
5/5 [==============================] - 3s 640ms/step - loss: 0.8633 - accuracy:
0.6353 - val_loss: 1.0385 - val_accuracy: 0.3800
Epoch 13/100
5/5 [==============================] - 3s 614ms/step - loss: 0.9190 - accuracy:
0.6291 - val_loss: 1.0355 - val_accuracy: 0.3800
Epoch 14/100
5/5 [==============================] - 3s 670ms/step - loss: 0.9265 - accuracy:
0.6190 - val_loss: 1.0369 - val_accuracy: 0.3800
Epoch 15/100
5/5 [==============================] - 3s 587ms/step - loss: 0.8580 - accuracy:
0.6348 - val_loss: 1.0338 - val_accuracy: 0.3833
Epoch 16/100
5/5 [==============================] - 3s 582ms/step - loss: 0.9122 - accuracy:
0.6219 - val_loss: 1.0405 - val_accuracy: 0.3767
Epoch 17/100
5/5 [==============================] - 2s 508ms/step - loss: 0.8447 - accuracy:
0.6580 - val_loss: 1.0361 - val_accuracy: 0.3933
Epoch 18/100
5/5 [==============================] - 3s 592ms/step - loss: 0.8573 - accuracy:
0.6552 - val_loss: 1.0360 - val_accuracy: 0.3900
Epoch 19/100
5/5 [==============================] - 3s 559ms/step - loss: 0.7839 - accuracy:
```

```
0.6789 - val_loss: 1.0291 - val_accuracy: 0.3900
Epoch 20/100
5/5 [==============================] - 3s 541ms/step - loss: 0.8119 - accuracy:
0.6665 - val_loss: 1.0157 - val_accuracy: 0.4133
Epoch 21/100
5/5 [==============================] - 3s 510ms/step - loss: 0.7916 - accuracy:
0.6721 - val_loss: 1.0256 - val_accuracy: 0.4233
Epoch 22/100
5/5 [==============================] - 3s 517ms/step - loss: 0.8218 - accuracy:
0.6633 - val_loss: 1.0302 - val_accuracy: 0.4267
Epoch 23/100
5/5 [==============================] - 3s 551ms/step - loss: 0.7875 - accuracy:
0.6722 - val_loss: 1.0384 - val_accuracy: 0.4333
Epoch 24/100
5/5 [==============================] - 3s 517ms/step - loss: 0.7512 - accuracy:
0.6949 - val_loss: 1.0470 - val_accuracy: 0.4300
Epoch 25/100
5/5 [==============================] - 3s 506ms/step - loss: 0.8116 - accuracy:
0.6815 - val_loss: 1.0571 - val_accuracy: 0.4167
Epoch 26/100
5/5 [==============================] - 3s 539ms/step - loss: 0.7220 - accuracy:
0.7134 - val_loss: 1.0560 - val_accuracy: 0.4200
Epoch 27/100
5/5 [==============================] - 3s 567ms/step - loss: 0.7797 - accuracy:
0.6843 - val_loss: 1.0754 - val_accuracy: 0.4200
Epoch 28/100
5/5 [==============================] - 2s 496ms/step - loss: 0.7558 - accuracy:
0.6947 - val_loss: 1.0797 - val_accuracy: 0.4233
Epoch 29/100
5/5 [==============================] - 3s 513ms/step - loss: 0.7365 - accuracy:
0.7218 - val_loss: 1.0851 - val_accuracy: 0.4300
Epoch 30/100
5/5 [==============================] - 3s 529ms/step - loss: 0.7185 - accuracy:
0.7143 - val_loss: 1.1049 - val_accuracy: 0.4200
Epoch 31/100
5/5 [==============================] - 3s 503ms/step - loss: 0.6940 - accuracy:
0.7319 - val_loss: 1.1508 - val_accuracy: 0.4067
Epoch 32/100
5/5 [==============================] - 3s 538ms/step - loss: 0.7057 - accuracy:
0.7338 - val_loss: 1.1706 - val_accuracy: 0.4067
Epoch 33/100
5/5 [==============================] - 3s 543ms/step - loss: 0.7211 - accuracy:
0.7184 - val_loss: 1.1485 - val_accuracy: 0.4133
Epoch 34/100
5/5 [==============================] - 3s 563ms/step - loss: 0.6585 - accuracy:
0.7610 - val_loss: 1.1447 - val_accuracy: 0.4200
Epoch 35/100
5/5 [==============================] - 3s 605ms/step - loss: 0.6765 - accuracy:
```

```
0.7269 - val_loss: 1.1458 - val_accuracy: 0.4400
Epoch 36/100
5/5 [==============================] - 3s 504ms/step - loss: 0.6581 - accuracy:
0.7220 - val_loss: 1.1863 - val_accuracy: 0.4233
Epoch 37/100
5/5 [==============================] - 2s 518ms/step - loss: 0.7127 - accuracy:
0.7220 - val_loss: 1.2255 - val_accuracy: 0.4100
Epoch 38/100
5/5 [==============================] - 3s 546ms/step - loss: 0.6691 - accuracy:
0.7427 - val_loss: 1.2278 - val_accuracy: 0.4167
Epoch 39/100
5/5 [==============================] - 3s 538ms/step - loss: 0.6896 - accuracy:
0.7282 - val_loss: 1.2267 - val_accuracy: 0.4167
Epoch 40/100
5/5 [==============================] - 3s 530ms/step - loss: 0.6475 - accuracy:
0.7549 - val_loss: 1.2413 - val_accuracy: 0.4067
Epoch 41/100
5/5 [==============================] - 3s 528ms/step - loss: 0.6532 - accuracy:
0.7481 - val_loss: 1.2510 - val_accuracy: 0.4067
Epoch 42/100
5/5 [==============================] - 3s 515ms/step - loss: 0.6088 - accuracy:
0.7537 - val_loss: 1.2479 - val_accuracy: 0.4100
Epoch 43/100
5/5 [==============================] - 3s 504ms/step - loss: 0.6496 - accuracy:
0.7688 - val_loss: 1.2300 - val_accuracy: 0.4100
Epoch 44/100
5/5 [==============================] - 3s 533ms/step - loss: 0.6674 - accuracy:
0.7324 - val_loss: 1.2054 - val_accuracy: 0.4333
Epoch 45/100
5/5 [==============================] - 2s 482ms/step - loss: 0.6580 - accuracy:
0.7447 - val_loss: 1.1793 - val_accuracy: 0.4367
Epoch 46/100
5/5 [==============================] - 3s 510ms/step - loss: 0.6250 - accuracy:
0.7453 - val_loss: 1.1880 - val_accuracy: 0.4500
Epoch 47/100
5/5 [==============================] - 2s 502ms/step - loss: 0.6368 - accuracy:
0.7396 - val_loss: 1.2120 - val_accuracy: 0.4433
Epoch 48/100
5/5 [==============================] - 3s 555ms/step - loss: 0.6315 - accuracy:
0.7516 - val_loss: 1.2117 - val_accuracy: 0.4433
Epoch 49/100
5/5 [==============================] - 3s 504ms/step - loss: 0.6516 - accuracy:
0.7438 - val_loss: 1.1991 - val_accuracy: 0.4533
Epoch 50/100
5/5 [==============================] - 3s 497ms/step - loss: 0.6606 - accuracy:
0.7460 - val_loss: 1.1837 - val_accuracy: 0.4467
Epoch 51/100
5/5 [==============================] - 2s 485ms/step - loss: 0.6217 - accuracy:
```

```
0.7418 - val_loss: 1.1753 - val_accuracy: 0.4333
Epoch 52/100
5/5 [==============================] - 3s 503ms/step - loss: 0.6561 - accuracy:
0.7436 - val_loss: 1.1610 - val_accuracy: 0.4333
Epoch 53/100
5/5 [==============================] - 2s 462ms/step - loss: 0.5816 - accuracy:
0.7685 - val_loss: 1.1128 - val_accuracy: 0.4500
Epoch 54/100
5/5 [==============================] - 3s 515ms/step - loss: 0.6159 - accuracy:
0.7600 - val_loss: 1.0466 - val_accuracy: 0.5033
Epoch 55/100
5/5 [==============================] - 3s 515ms/step - loss: 0.6039 - accuracy:
0.7709 - val_loss: 1.0186 - val_accuracy: 0.5167
Epoch 56/100
5/5 [==============================] - 2s 476ms/step - loss: 0.5752 - accuracy:
0.7844 - val_loss: 1.0032 - val_accuracy: 0.5367
Epoch 57/100
5/5 [==============================] - 3s 589ms/step - loss: 0.5657 - accuracy:
0.7770 - val_loss: 0.9961 - val_accuracy: 0.5367
Epoch 58/100
5/5 [==============================] - 2s 470ms/step - loss: 0.5510 - accuracy:
0.7787 - val_loss: 1.0059 - val_accuracy: 0.5367
Epoch 59/100
5/5 [==============================] - 2s 474ms/step - loss: 0.5863 - accuracy:
0.7667 - val_loss: 1.0071 - val_accuracy: 0.5600
Epoch 60/100
5/5 [==============================] - 3s 534ms/step - loss: 0.5927 - accuracy:
0.7727 - val_loss: 1.0120 - val_accuracy: 0.5667
Epoch 61/100
5/5 [==============================] - 2s 459ms/step - loss: 0.5751 - accuracy:
0.7668 - val_loss: 0.9779 - val_accuracy: 0.5900
Epoch 62/100
5/5 [==============================] - 3s 542ms/step - loss: 0.6055 - accuracy:
0.7748 - val_loss: 0.9260 - val_accuracy: 0.6000
Epoch 63/100
5/5 [==============================] - 3s 499ms/step - loss: 0.6096 - accuracy:
0.7730 - val_loss: 0.9117 - val_accuracy: 0.6033
Epoch 64/100
5/5 [==============================] - 2s 508ms/step - loss: 0.6054 - accuracy:
0.7542 - val_loss: 0.9234 - val_accuracy: 0.6000
Epoch 65/100
5/5 [==============================] - 2s 429ms/step - loss: 0.5709 - accuracy:
0.7893 - val_loss: 0.9346 - val_accuracy: 0.6033
Epoch 66/100
5/5 [==============================] - 2s 458ms/step - loss: 0.5365 - accuracy:
0.7871 - val_loss: 0.9332 - val_accuracy: 0.6000
Epoch 67/100
5/5 [==============================] - 2s 478ms/step - loss: 0.5671 - accuracy:
```

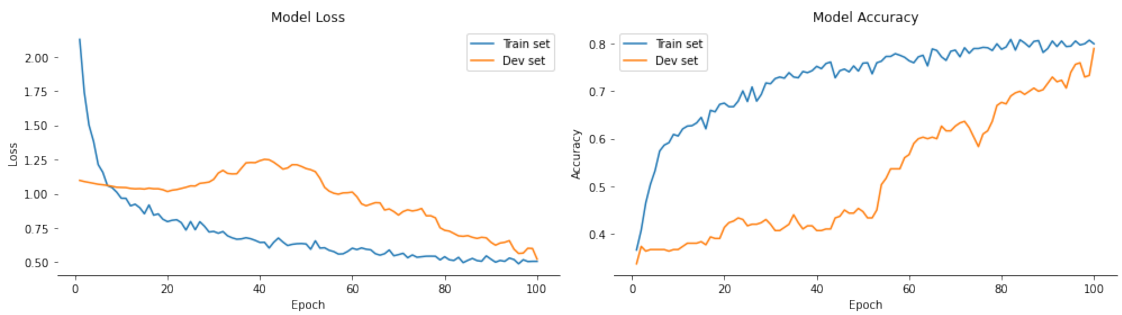
```
0.7677 - val_loss: 0.8807 - val_accuracy: 0.6267
Epoch 68/100
5/5 [==============================] - 3s 540ms/step - loss: 0.5923 - accuracy:
0.7657 - val_loss: 0.8887 - val_accuracy: 0.6167
Epoch 69/100
5/5 [==============================] - 3s 601ms/step - loss: 0.5350 - accuracy:
0.7984 - val_loss: 0.8692 - val_accuracy: 0.6167
Epoch 70/100
5/5 [==============================] - 3s 534ms/step - loss: 0.5421 - accuracy:
0.7985 - val_loss: 0.8438 - val_accuracy: 0.6267
Epoch 71/100
5/5 [==============================] - 2s 492ms/step - loss: 0.5536 - accuracy:
0.7810 - val_loss: 0.8695 - val_accuracy: 0.6333
Epoch 72/100
5/5 [==============================] - 2s 465ms/step - loss: 0.5404 - accuracy:
0.7923 - val_loss: 0.8823 - val_accuracy: 0.6367
Epoch 73/100
5/5 [==============================] - 2s 455ms/step - loss: 0.5329 - accuracy:
0.7811 - val_loss: 0.8737 - val_accuracy: 0.6233
Epoch 74/100
5/5 [==============================] - 3s 516ms/step - loss: 0.5428 - accuracy:
0.7851 - val_loss: 0.8797 - val_accuracy: 0.6033
Epoch 75/100
5/5 [==============================] - 3s 556ms/step - loss: 0.5375 - accuracy:
0.7924 - val_loss: 0.8919 - val_accuracy: 0.5833
Epoch 76/100
5/5 [==============================] - 3s 532ms/step - loss: 0.5285 - accuracy:
0.8018 - val_loss: 0.8386 - val_accuracy: 0.6100
Epoch 77/100
5/5 [==============================] - 3s 530ms/step - loss: 0.5477 - accuracy:
0.7877 - val_loss: 0.8388 - val_accuracy: 0.6167
Epoch 78/100
5/5 [==============================] - 3s 472ms/step - loss: 0.5468 - accuracy:
0.7751 - val_loss: 0.8235 - val_accuracy: 0.6367
Epoch 79/100
5/5 [==============================] - 2s 497ms/step - loss: 0.5184 - accuracy:
0.7925 - val_loss: 0.7515 - val_accuracy: 0.6700
Epoch 80/100
5/5 [==============================] - 3s 538ms/step - loss: 0.5588 - accuracy:
0.7805 - val_loss: 0.7325 - val_accuracy: 0.6767
Epoch 81/100
5/5 [==============================] - 2s 478ms/step - loss: 0.5300 - accuracy:
0.7936 - val_loss: 0.7266 - val_accuracy: 0.6733
Epoch 82/100
5/5 [==============================] - 2s 512ms/step - loss: 0.5223 - accuracy:
0.7995 - val_loss: 0.7094 - val_accuracy: 0.6900
Epoch 83/100
5/5 [==============================] - 2s 492ms/step - loss: 0.5289 - accuracy:
```

```
0.7895 - val_loss: 0.6918 - val_accuracy: 0.6967
Epoch 84/100
5/5 [==============================] - 3s 531ms/step - loss: 0.4981 - accuracy:
0.8154 - val_loss: 0.6886 - val_accuracy: 0.7000
Epoch 85/100
5/5 [==============================] - 2s 504ms/step - loss: 0.5132 - accuracy:
0.8044 - val_loss: 0.6937 - val_accuracy: 0.6933
Epoch 86/100
5/5 [==============================] - 3s 487ms/step - loss: 0.5553 - accuracy:
0.7898 - val_loss: 0.6813 - val_accuracy: 0.7000
Epoch 87/100
5/5 [==============================] - 3s 516ms/step - loss: 0.4799 - accuracy:
0.8224 - val_loss: 0.6732 - val_accuracy: 0.7067
Epoch 88/100
5/5 [==============================] - 3s 535ms/step - loss: 0.5195 - accuracy:
0.8072 - val_loss: 0.6814 - val_accuracy: 0.7000
Epoch 89/100
5/5 [==============================] - 2s 468ms/step - loss: 0.5355 - accuracy:
0.7856 - val_loss: 0.6775 - val_accuracy: 0.7033
Epoch 90/100
5/5 [==============================] - 3s 541ms/step - loss: 0.5146 - accuracy:
0.7965 - val_loss: 0.6455 - val_accuracy: 0.7167
Epoch 91/100
5/5 [==============================] - 2s 507ms/step - loss: 0.4927 - accuracy:
0.8126 - val_loss: 0.6237 - val_accuracy: 0.7300
Epoch 92/100
5/5 [==============================] - 3s 508ms/step - loss: 0.4888 - accuracy:
0.8118 - val_loss: 0.6396 - val_accuracy: 0.7200
Epoch 93/100
5/5 [==============================] - 3s 520ms/step - loss: 0.5181 - accuracy:
0.8045 - val_loss: 0.6450 - val_accuracy: 0.7233
Epoch 94/100
5/5 [==============================] - 3s 603ms/step - loss: 0.5242 - accuracy:
0.7979 - val_loss: 0.6570 - val_accuracy: 0.7067
Epoch 95/100
5/5 [==============================] - 3s 576ms/step - loss: 0.5033 - accuracy:
0.7990 - val_loss: 0.5948 - val_accuracy: 0.7400
Epoch 96/100
5/5 [==============================] - 3s 493ms/step - loss: 0.4799 - accuracy:
0.8098 - val_loss: 0.5635 - val_accuracy: 0.7567
Epoch 97/100
5/5 [==============================] - 3s 524ms/step - loss: 0.5402 - accuracy:
0.7871 - val_loss: 0.5662 - val_accuracy: 0.7600
Epoch 98/100
5/5 [==============================] - 3s 568ms/step - loss: 0.4984 - accuracy:
0.8055 - val_loss: 0.6014 - val_accuracy: 0.7300
Epoch 99/100
5/5 [==============================] - 3s 597ms/step - loss: 0.4773 - accuracy:
```

```
0.8159 - val_loss: 0.5996 - val_accuracy: 0.7333
Epoch 100/100
5/5 [==============================] - 3s 547ms/step - loss: 0.5104 - accuracy:
0.8024 - val_loss: 0.5223 - val_accuracy: 0.7900
```

1.4.2 Prediction and Performance Analysis

[21]:
```
plot_loss_and_accuracy_am2(history=history_cnn)
```



Let's evaluate the performance of this model under unseen data (x_test)

[22]:
```
model_cnn.evaluate(x_test_red,y_test_oh)

loss_value_cnn, acc_value_cnn = model_cnn.evaluate(x_test_red, y_test_oh,␣
↪verbose=0)
print('Loss value: ', loss_value_cnn)
print('Acurracy value: ', acc_value_cnn)
```

```
47/47 [==============================] - 0s 7ms/step - loss: 0.6509 - accuracy:
0.7380
Loss value:  0.6508954167366028
Acurracy value:  0.7379999756813049
```

Task: Discuss CNN and MLP results.

Your Turn: Now we changed our mind, we found that detecting airplanes, horses and trucks is a bit boring :(. We would like to detect whether an image has a bird, a dog or a ship =)

Implement a CNN to classify the images of the new reduced dataset.

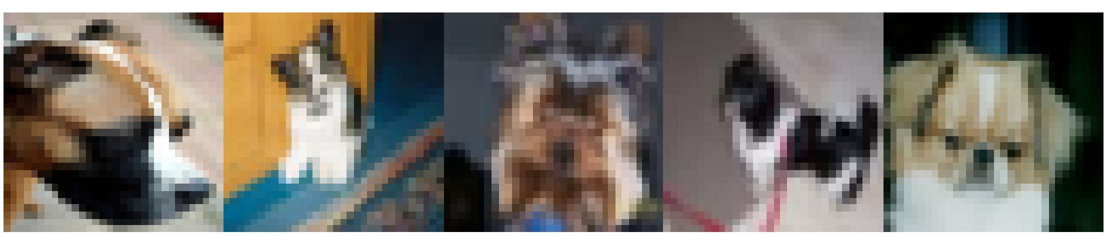
Creating the dataset

[23]:
```
# Lets select just 3 classes to make this tutorial feasible
selected_idx = np.array([2, 5, 8])
n_images = 1500

y_train_idx = np.isin(y_train, selected_idx)
y_test_idx = np.isin(y_test, selected_idx)
```

```
y_train_new = y_train[y_train_idx][:n_images]
x_train_new = x_train[y_train_idx][:n_images]

y_test_new = y_test[y_test_idx][:n_images]
x_test_new = x_test[y_test_idx][:n_images]

# replacing the labels 0, 7 and 9 to 0, 1, 2 repectively.
y_train_new[y_train_new == selected_idx[0]] = 0
y_train_new[y_train_new == selected_idx[1]] = 1
y_train_new[y_train_new == selected_idx[2]] = 2

y_test_new[y_test_new == selected_idx[0]] = 0
y_test_new[y_test_new == selected_idx[1]] = 1
y_test_new[y_test_new == selected_idx[2]] = 2

# visulising the images in the reduced dataset
plot_samples(x_train_new, y_train_new, class_name[selected_idx])
```

```
--------------------------------------------------------------------------------
------------

bird - number of samples: 513
```

```
--------------------------------------------------------------------------------
------------

dog - number of samples: 480
```



```
--------------------------------------------------------------------------------
------------
ship - number of samples: 507
```

Pre-processing the new dataset

[24]:
```python
# normalising the data
x_train_new = x_train_new.astype('float32')
x_test_new = x_test_new.astype('float32')
x_train_new /= 255.0
x_test_new /= 255.0

# creating the one-hot representation
y_train_oh_n = keras.utils.to_categorical(y_train_new)
y_test_oh_n = keras.utils.to_categorical(y_test_new)

print('Label: ',y_train_new[0], ' one-hot: ', y_train_oh_n[0])
print('Label: ',y_train_new[810], ' one-hot: ',  y_train_oh_n[810])
print('Label: ',y_test_new[20], ' one-hot: ', y_test_oh_n[20])
```

```
Label:  0  one-hot:  [1. 0. 0.]
Label:  1  one-hot:  [0. 1. 0.]
Label:  2  one-hot:  [0. 0. 1.]
```

Step 1: Create the CNN Model.

For example, you can try (Danger, Will Robinson! This model can overfits):

```python
model_cnn_new = Sequential()

model_cnn_new.add(Conv2D(32, (3, 3), padding='valid', activation = 'relu',
                 input_shape=x_train_new.shape[1:]))
model_cnn_new.add(BatchNormalization())
model_cnn_new.add(MaxPooling2D(pool_size=(2,2)))
model_cnn_new.add(Dropout(0.7))

# You can stack several convolution layers before apply BatchNormalization, MaxPooling2D
# and Dropout
model_cnn_new.add(Conv2D(32, (3, 3), padding='valid', activation = 'relu',
```

```
                 input_shape=x_train_new.shape[1:]))
model_cnn_new.add(Conv2D(16, (3, 3), padding='valid', activation = 'relu'))
model_cnn_new.add(Conv2D(64, (3, 3), padding='valid', activation = 'relu'))
model_cnn_new.add(BatchNormalization())
# You can also don't use max pooling... it is up to you
#model_cnn_new.add(MaxPooling2D(pool_size=(2,2)))  # this line can lead to negative dimension 
model_cnn_new.add(Dropout(0.7))

model_cnn_new.add(Conv2D(32, (5, 5), padding='valid', activation = 'relu'))
model_cnn_new.add(BatchNormalization())
model_cnn_new.add(MaxPooling2D(pool_size=(2,2)))
model_cnn_new.add(Dropout(0.7))

model_cnn_new.add(Flatten())
model_cnn_new.add(Dense(128, activation = 'relu'))

model_cnn_new.add(Dense(y_test_oh_n.shape[1], activation='softmax'))
```

[ ]:

```

```

Step 2: Summarise the model.

For example, you can try:

```
model_cnn_new.summary()
```

[ ]:

```

```

Step 3: Define optimiser (try 'rmsprop', 'sgd', 'adagrad' or 'adadelta' if you wich), loss and metric

For example:

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

[ ]:

```

```

Step 4: Train the model, here you can define the number of epochs and batch_size that best fit for you model

For example:

```
# this can take SEVERAL minutes or even hours.. days... if your model is quite deep
history_cnn_new = model_cnn_new.fit(x_train_new,
                    y_train_oh_n,
                    batch_size = 256,
                    epochs = 100,
                    validation_split = 0.2,
                    verbose = 1)
```

```
[ ]:
```

Step 4: Evaluate the model performance by using the metric that you think is the best.

For example:

```
model_cnn_new.evaluate(x_test_new,y_test_oh_n)

loss_value_cnn_n, acc_value_cnn_n = model_cnn_new.evaluate(x_test_new, y_test_oh_n, verbose=0)
print('Loss value: ', loss_value_cnn_n)
print('Acurracy value: ', acc_value_cnn_n)
```

Plot the loss and accuracy if you which.

```
[ ]:
```

```
[ ]:
```