CNN Image Classification Laboration

Images used in this laboration are from CIFAR 10 (https://en.wikipedia.org/wiki/CIFAR-10). The CIFAR-10 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.

You need to answer all questions in this notebook.

Part 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.

https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.convolve2d.html

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).

In [28]:

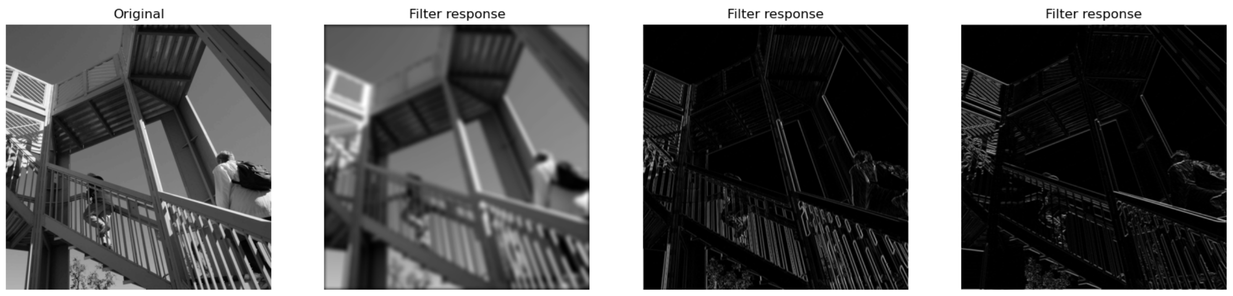
```python
# This cell is finished

from scipy import signal
import numpy as np

# Get a test image
from scipy import misc
image = misc.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]])
```

```
C:\Users\Dell\AppData\Local\Temp\ipykernel_15876\2994295117.py:8: DeprecationWarning:
scipy.misc.ascent has been deprecated in SciPy v1.10.0; and will be completely remove
d in SciPy v1.12.0. Dataset methods have moved into the scipy.datasets module. Use sc
ipy.datasets.ascent instead.
  image = misc.ascent()
```

In [29]:
```
# Perform convolution using the function 'convolve2d' for the different filters
filterResponseGauss = signal.convolve2d(image, gaussFilter, boundary='fill', mode='sar
filterResponseSobelX = signal.convolve2d(image,sobelX , boundary='fill', mode='same')
filterResponseSobelY = signal.convolve2d(image,sobelY, boundary='fill', mode='same')
```

In [30]:
```
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('Filter response')
ax_filt1.set_axis_off()
ax_filt2.imshow(np.absolute(filterResponseSobelX), cmap='gray')
ax_filt2.set_title('Filter response')
ax_filt2.set_axis_off()
ax_filt3.imshow(np.absolute(filterResponseSobelY), cmap='gray')
ax_filt3.set_title('Filter response')
ax_filt3.set_axis_off()
```



Part 2: Understanding convolutions

Question 1: What do the 3 different filters (Gaussian, SobelX, SobelY) do to the original image?

Answer:

- Gaussian filter: This filter applies a blurring effect to the image, which can help to reduce noise and make edges and details appear smoother. The specific blurring effect depends on the size and standard deviation of the Gaussian filter used.

- SobelX filter: This filter is designed to detect horizontal edges in an image by highlighting regions where the pixel values change rapidly along the x-axis.
- SobelY filter: This filter is similar to the SobelX filter, but it is designed to detect vertical edges in an image by highlighting regions where the pixel values change rapidly along the y-axis.

Question 2: What is the size of the original image? How many channels does it have? How many channels does a color image normally have?

Answer:

The size of the original image is 512 X 512. It has only one channel because it is a greyscale image. A color image has three channels (red, green, and blue), which together form the full color spectrum of the image. Question 3: What is the size of the different filters?

Answer:

The Gaussian filter has a size of 15 x 15. The Sobel X and Sobel Y filters are of size 3 x 3.

Question 4: What is the size of the filter response if mode 'same' is used for the convolution ?

Answer:

If mode='same' is used in convolution, then the output size of the filtered image will be the same as the input image. The filtered image will be cropped at the edges to match the size of the input image.

Question 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?

Answer:

When using the "valid" mode for convolution, only the parts of the image where the filter can completely overlap with are used for the convolution. As a result, the output will be smaller than the input, not the other way around. The size of output will be smaller than the input size by an amount equal to the size of the filter minus one in each dimension.

Question 6: Why are 'valid' convolutions a problem for CNNs with many layers?

Answer:

Valid convolutions can be a problem for CNNs with many layers because the output size of the convolutional layer becomes smaller and smaller with each layer. This can cause a loss of information because the spatial resolution of the input data is reduced. If the output size becomes too small, it may not be possible to distinguish between different features in the image, and the performance of the model may become poor.

In [31]:
```
# Your code for checking sizes of image and filter responses

print(f"Size of the original image {image.shape}")
print(f"Size of the image with gaussian filter response {filterResponseGauss.shape}")
print(f"Size of the image with SobelX filter response {filterResponseSobelX.shape}")
print(f"Size of the image with SobelX filter response {filterResponseSobelY.shape}")
```

```
Size of the original image (512, 512)
Size of the image with gaussian filter response (512, 512)
Size of the image with SobelX filter response (512, 512)
Size of the image with SobelX filter response (512, 512)
```

Part 3: Get a graphics card

Skip this part if you run on a CPU (recommended)

Let's make sure that our script can see the graphics card that will be used. The graphics cards will perform all the time consuming convolutions in every training iteration.

In [ ]:

```
import os
import warnings

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

import keras.backend as K
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";

# Allow growth of GPU memory, otherwise it will always look like all the memory is bei
physical_devices = tf.config.experimental.list_physical_devices('GPU')
tf.config.experimental.set_memory_growth(physical_devices[0], True)
```

Part 4: How fast is the graphics card?

Question 7: Why are the filters used for a color image of size 7 x 7 x 3, and not 7 x 7 ?

Answer:

Color images are represented in 3 channels: red, green, and blue. So, when applying a filter to a color image, we have to consider all three channels, not just one. This is reason that the filters used for a color image are of size 7 x 7 x 3.

Question 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?

Answer:

The 'Conv2D' layer performs convolution operation on input data using a filter.The filter is slid over the input data, multiplying and summing the results to produce an output feature map. Filter parameters are learned during training to extract useful features for a given task like image classification.

Yes, the operation performed by the 'Conv2D' layer in a convolutional neural network is a standard 2D convolution, similar to the one performed by the function signal.convolve2d that we tested. Question 9: 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.

Answer:

When convolving a batch of images, the workload can be distributed across the available processing cores, allowing the GPU to perform the task much faster than a CPU. Therefore, whether we convolve a batch of 1,000 images or 3 images, a GPU will be faster than a CPU due to its parallel computing capability.

Part 5: Load data

Time to make a 2D CNN. Load the images and labels from keras.datasets, this cell is already finished.

In [32]:
```
from keras.datasets import cifar10
import numpy as np

classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'tru

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

print("Training images have size {} and labels have size {} ".format(Xtrain.shape, Ytr
print("Test images have size {} and labels have size {} \n ".format(Xtest.shape, Ytest

# Reduce the number of images for training and testing to 10000 and 2000 respectively,
# to reduce processing time for this laboration
Xtrain = Xtrain[0:10000]
Ytrain = Ytrain[0:10000]

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

Ytestint = Ytest

print("Reduced training images have size %s and labels have size %s " % (Xtrain.shape,
print("Reduced test images have size %s and labels have size %s \n" % (Xtest.shape, Yt

# Check that we have some training examples from each class
for i in range(10):
    print("Number of training examples for class {} is {}" .format(i,np.sum(Ytrain ==
Training 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 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 examples for class 0 is 1005
Number of training examples for class 1 is 974
Number of training examples for class 2 is 1032
Number of training examples for class 3 is 1016
Number of training examples for class 4 is 999
Number of training examples for class 5 is 937
Number of training examples for class 6 is 1030
Number of training examples for class 7 is 1001
Number of training examples for class 8 is 1025
Number of training examples for class 9 is 981
```

Part 6: Plotting

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

In [33]:
```python
import matplotlib.pyplot as plt

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

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

Part 7: Split data into training, validation and testing

Split your training data into training (Xtrain, Ytrain) and validation (Xval, Yval), so that we have training, validation and test datasets (as in the previous laboration). We use a function in scikit learn. Use 25% of the data for validation.

https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html

In [34]:
```
from sklearn.model_selection import train_test_split

# Your code for splitting the dataset
Xtrain, Xval, Ytrain, Yval = train_test_split(Xtrain, Ytrain, test_size=0.25,random_st
# Print the size of training data, validation data and test data
print(f"Size of training data :{Xtrain.shape}")
print(f"Size of labels of training data :{Ytrain.shape}")
print(f"Size of validation data :{Xval.shape}")
print(f"Size of labels for validation data :{Yval.shape}")
print(f"Size of test data :{Xtest.shape}")
print(f"Size of labels for test data :{Ytest.shape}")
```

```
Size of training data :(7500, 32, 32, 3)
Size of labels of training data :(7500, 1)
Size of validation data :(2500, 32, 32, 3)
Size of labels for validation data :(2500, 1)
Size of test data :(2000, 32, 32, 3)
Size of labels for test data :(2000, 1)
```

Part 8: Preprocessing of images

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. This cell is already finished.

In [35]:
```
# 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
```

Part 9: Preprocessing of labels

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 a function in Keras, see https://keras.io/api/utils/python_utils/#to_categorical-function

In [36]:
```
from tensorflow.keras.utils import to_categorical

# Print shapes before converting the labels
print(f"Size of Ytrain :{Ytrain.shape}")
print(f"Size of Yval :{Yval.shape}")
print(f"Size of Ytest :{Ytest.shape}")

# Your code for converting Ytrain, Yval, Ytest to categorical
Ytrain = to_categorical(Ytrain,dtype="int32")
Yval = to_categorical(Yval,dtype="int32")
```

```
Ytest = to_categorical(Ytest,dtype="int32")

# Print shapes after converting the labels
print(f"Size of Ytrain :{Ytrain.shape}")
print(f"Size of Yval :{Yval.shape}")
print(f"Size of Ytest :{Ytest.shape}")
```

```
Size of Ytrain :(7500, 1)
Size of Yval :(2500, 1)
Size of Ytest :(2000, 1)
Size of Ytrain :(7500, 10)
Size of Yval :(2500, 10)
Size of Ytest :(2000, 10)
```

Part 10: 2D CNN

Finish this code to create the image classifier, using a 2D CNN. Each convolutional layer will contain 2D convolution, batch normalization and max pooling. After the convolutional layers comes a flatten layer and a number of intermediate dense layers. The convolutional layers should take the number of filters as an argument, use a kernel size of 3 x 3, 'same' padding, and relu activation functions. The number of filters will double with each convolutional layer. The max pooling layers should have a pool size of 2 x 2. The intermediate dense layers before the final dense layer should take the number of nodes as an argument, use relu activation functions, and be followed by batch normalization. The final dense layer should have 10 nodes (= the number of classes in this laboration) and 'softmax' activation. Here we start with the Adam optimizer.

Relevant functions are

`model.add()` , adds a layer to the network

`Dense()` , a dense network layer

`Conv2D()` , performs 2D convolutions with a number of filters with a certain size (e.g. 3 x 3).

`BatchNormalization()` , perform batch normalization

`MaxPooling2D()` , saves the max for a given pool size, results in down sampling

`Flatten()` , flatten a multi-channel tensor into a long vector

`model.compile()` , compile the model, add " metrics=['accuracy'] " to print the classification accuracy during the training

See https://keras.io/api/layers/core_layers/dense/ and https://keras.io/api/layers/reshaping_layers/flatten/ for information on how the `Dense()` and `Flatten()` functions work

See https://keras.io/layers/convolutional/ for information on how `Conv2D()` works

See https://keras.io/layers/pooling/ for information on how `MaxPooling2D()` works

Import a relevant cost function for multi-class classification from keras.losses (https://keras.io/losses/) , it relates to how many classes you have.

See the following links for how to compile, train and evaluate the model

https://keras.io/api/models/model_training_apis/#compile-method

https://keras.io/api/models/model_training_apis/#fit-method

https://keras.io/api/models/model_training_apis/#evaluate-method

In [37]:
```
from keras.models import Sequential, Model
from keras.layers import Input, Conv2D, BatchNormalization, MaxPooling2D, Flatten, De
from tensorflow.keras.optimizers import Adam
from keras.losses import categorical_crossentropy

# Set seed from random number generator, for better comparisons
from numpy.random import seed
seed(123)

def build_CNN(input_shape, n_conv_layers=2, n_filters=16, n_dense_layers=0, n_nodes=5

    # Setup a sequential model
    model = Sequential()

    # Add first convolutional layer to the model, requires input shape
    model.add( Conv2D(filters=n_filters, kernel_size = (3, 3),activation ='relu', inpu
    model.add(MaxPooling2D(pool_size=(2,2)))

    # Add remaining convolutional layers to the model, the number of filters should i
    for i in range(n_conv_layers-1):
        model.add( Conv2D(filters=2**(i+1)*n_filters,kernel_size = (3, 3),activation =
        model.add(MaxPooling2D(pool_size=(2,2)))

    # Add flatten layer
    model.add(Flatten())

    # Add intermediate dense layers
    for i in range(n_dense_layers):
        model.add(Dense(n_nodes, activation = 'relu'))
        if use_dropout is True:
            model.add(Dropout(0.5))

    # Add final dense layer
    model.add(Dense(10, activation = 'softmax'))

    # Compile model
    model.compile(loss=categorical_crossentropy, optimizer=Adam(learning_rate=learnin

    return model
```

In [38]:
```
# Lets define a help function for plotting the training results
import matplotlib.pyplot as plt
def plot_results(history):

    loss = history.history['loss']
    acc = history.history['accuracy']
    val_loss = history.history['val_loss']
    val_acc = history.history['val_accuracy']

    plt.figure(figsize=(10,4))
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.plot(loss)
    plt.plot(val_loss)
    plt.legend(['Training','Validation'])

    plt.figure(figsize=(10,4))
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.plot(acc)
    plt.plot(val_acc)
    plt.legend(['Training','Validation'])

    plt.show()
```

Part 11: Train 2D CNN

Time to train the 2D CNN, start with 2 convolutional layers, no intermediate dense layers, learning rate = 0.01. The first convolutional layer should have 16 filters (which means that the second convolutional layer will have 32 filters).

Relevant functions

`build_CNN` , the function we defined in Part 10, call it with the parameters you want to use

`model.fit()` , train the model with some training data

`model.evaluate()` , apply the trained model to some test data

See the following links for how to train and evaluate the model

https://keras.io/api/models/model_training_apis/#fit-method

https://keras.io/api/models/model_training_apis/#evaluate-method

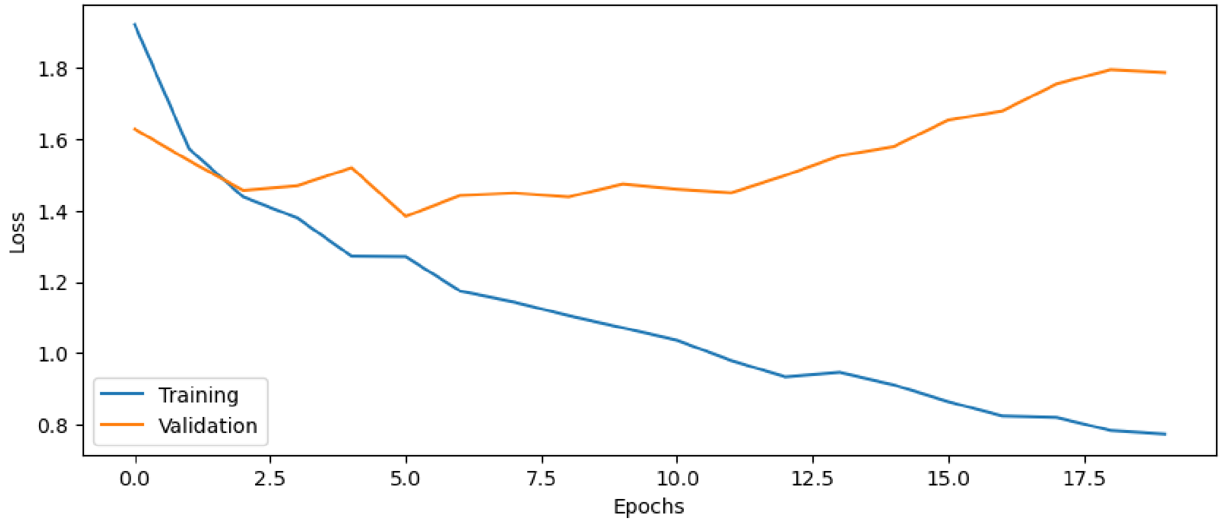
2 convolutional layers, no intermediate dense layers

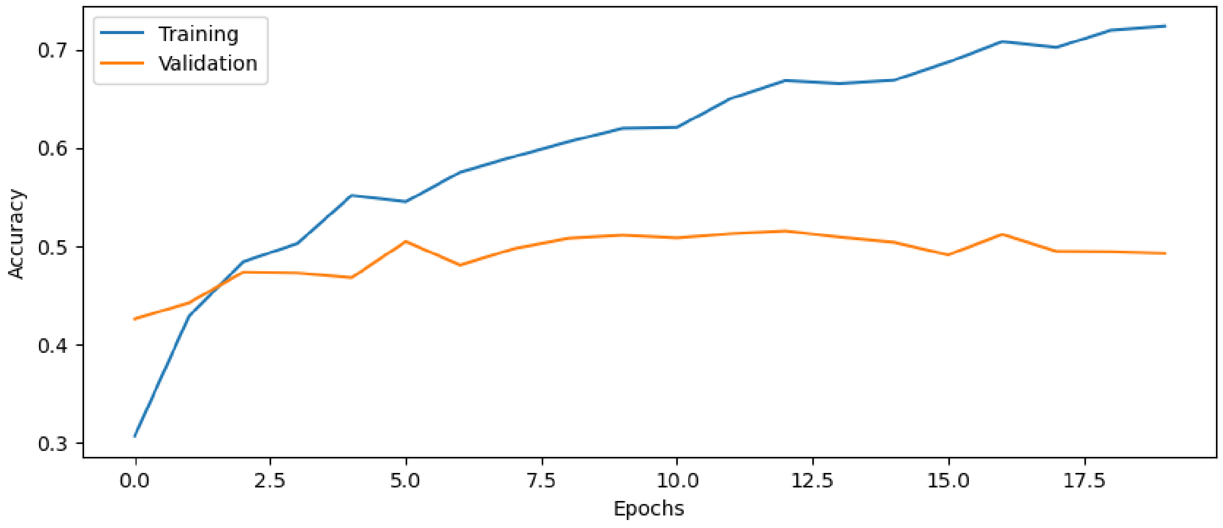
In [39]:
```
# Setup some training parameters
batch_size = 100
epochs = 20
input_shape = Xtrain.shape[1:4]

# Build model
model1 = build_CNN(input_shape, n_conv_layers=2, n_filters=16, n_dense_layers=0, n_noc

# Train the model  using training data and validation data
history1 = model1.fit(Xtrain,Ytrain, batch_size=batch_size,epochs=epochs, verbose=1, \
Epoch 1/20
75/75 [==============================] - 9s 74ms/step - loss: 1.9213 - accuracy: 0.30
64 - val_loss: 1.6285 - val_accuracy: 0.4256
Epoch 2/20
75/75 [==============================] - 3s 34ms/step - loss: 1.5733 - accuracy: 0.42
84 - val_loss: 1.5399 - val_accuracy: 0.4420
Epoch 3/20
75/75 [==============================] - 3s 35ms/step - loss: 1.4382 - accuracy: 0.48
37 - val_loss: 1.4566 - val_accuracy: 0.4732
Epoch 4/20
75/75 [==============================] - 3s 41ms/step - loss: 1.3789 - accuracy: 0.50
24 - val_loss: 1.4698 - val_accuracy: 0.4724
Epoch 5/20
75/75 [==============================] - 3s 35ms/step - loss: 1.2717 - accuracy: 0.55
13 - val_loss: 1.5199 - val_accuracy: 0.4676
Epoch 6/20
75/75 [==============================] - 3s 39ms/step - loss: 1.2704 - accuracy: 0.54
49 - val_loss: 1.3836 - val_accuracy: 0.5044
Epoch 7/20
75/75 [==============================] - 3s 37ms/step - loss: 1.1741 - accuracy: 0.57
49 - val_loss: 1.4422 - val_accuracy: 0.4804
Epoch 8/20
75/75 [==============================] - 3s 35ms/step - loss: 1.1430 - accuracy: 0.59
09 - val_loss: 1.4491 - val_accuracy: 0.4972
Epoch 9/20
75/75 [==============================] - 3s 36ms/step - loss: 1.1050 - accuracy: 0.60
60 - val_loss: 1.4380 - val_accuracy: 0.5076
Epoch 10/20
75/75 [==============================] - 3s 35ms/step - loss: 1.0709 - accuracy: 0.61
97 - val_loss: 1.4743 - val_accuracy: 0.5108
Epoch 11/20
75/75 [==============================] - 3s 38ms/step - loss: 1.0361 - accuracy: 0.62
07 - val_loss: 1.4600 - val_accuracy: 0.5080
Epoch 12/20
75/75 [==============================] - 3s 35ms/step - loss: 0.9793 - accuracy: 0.65
01 - val_loss: 1.4498 - val_accuracy: 0.5124
Epoch 13/20
75/75 [==============================] - 3s 35ms/step - loss: 0.9333 - accuracy: 0.66
81 - val_loss: 1.4984 - val_accuracy: 0.5152
Epoch 14/20
75/75 [==============================] - 3s 34ms/step - loss: 0.9459 - accuracy: 0.66
51 - val_loss: 1.5532 - val_accuracy: 0.5088
Epoch 15/20
75/75 [==============================] - 3s 36ms/step - loss: 0.9108 - accuracy: 0.66
85 - val_loss: 1.5794 - val_accuracy: 0.5036
Epoch 16/20
75/75 [==============================] - 3s 34ms/step - loss: 0.8639 - accuracy: 0.68
69 - val_loss: 1.6534 - val_accuracy: 0.4908
Epoch 17/20
75/75 [==============================] - 3s 37ms/step - loss: 0.8237 - accuracy: 0.70
79 - val_loss: 1.6793 - val_accuracy: 0.5116
Epoch 18/20
75/75 [==============================] - 3s 36ms/step - loss: 0.8198 - accuracy: 0.70
20 - val_loss: 1.7548 - val_accuracy: 0.4944
Epoch 19/20
75/75 [==============================] - 3s 35ms/step - loss: 0.7828 - accuracy: 0.71
95 - val_loss: 1.7953 - val_accuracy: 0.4940
Epoch 20/20
75/75 [==============================] - 3s 36ms/step - loss: 0.7728 - accuracy: 0.72
36 - val_loss: 1.7872 - val_accuracy: 0.4924
```

In [40]:
```
# Evaluate the trained model on test set, not used in training or validation
score = model1.evaluate(Xtest, Ytest, verbose=1)
print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])
```

```
63/63 [==============================] - 2s 23ms/step - loss: 1.8405 - accuracy: 0.48
90
Test loss: 1.8405
Test accuracy: 0.4890
```

In [41]:
```
# Plot the history from the training run
plot_results(history1)
```





Part 12: Improving performance

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

Question 10: How big is the difference between training and test accuracy?

Answer:

Training accuracy is 72.36% and test accuracy is 48.90%. The difference is 23.46%

Question 11: For the DNN laboration we used a batch size of 10,000, why do we need to use a smaller batch size in this laboration?

Answer:

In CNN, the input data goes through several convolutional layers, which apply filters to the data and produce output feature maps. The number of parameters in these filters can be quite large, which makes training with a large batch size.

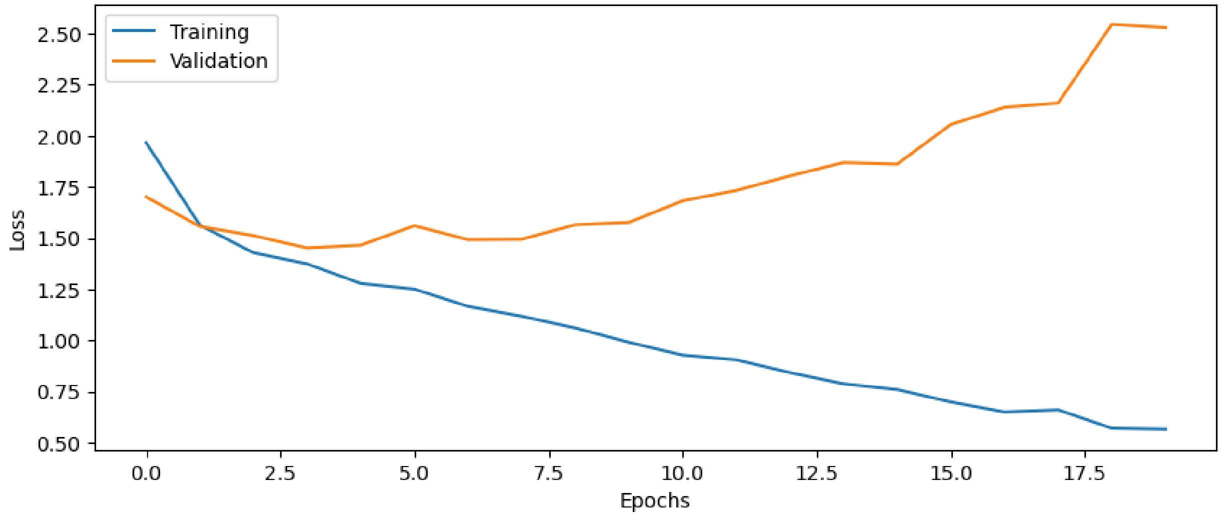
2 convolutional layers, 1 intermediate dense layer (50 nodes)

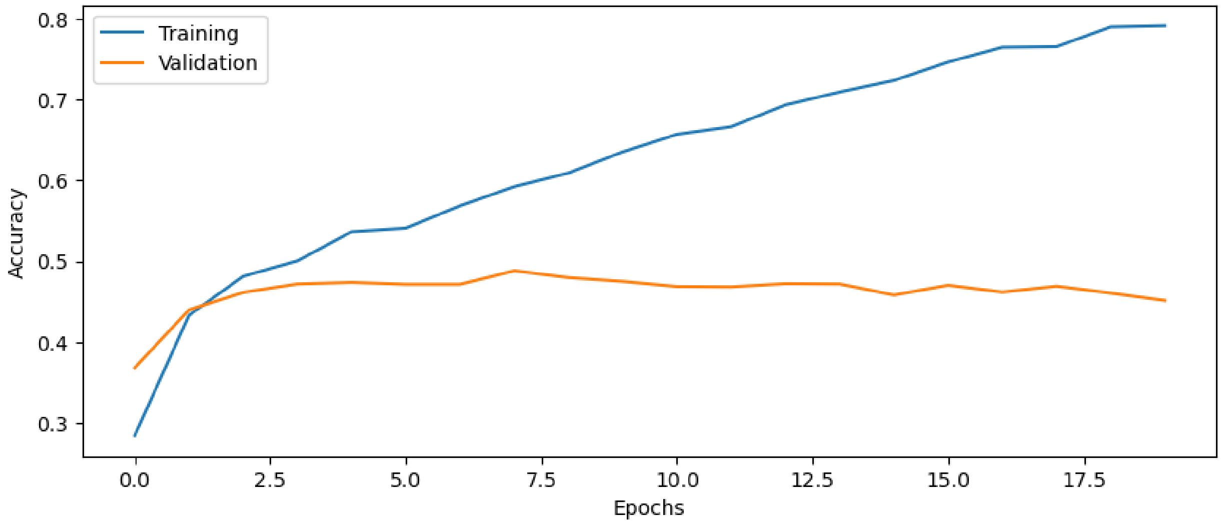
In [43]:
```
# Setup some training parameters
batch_size = 100
epochs = 20
input_shape = Xtrain.shape[1:4]

# Build model
model2 = build_CNN(input_shape, n_conv_layers=2, n_filters=16, n_dense_layers=1, n_noc

# Train the model  using training data and validation data
history2 = model2.fit(Xtrain,Ytrain, batch_size=batch_size,epochs=epochs, verbose=1, v
Epoch 1/20
75/75 [==============================] - 10s 76ms/step - loss: 1.9663 - accuracy: 0.2
841 - val_loss: 1.7013 - val_accuracy: 0.3680
Epoch 2/20
75/75 [==============================] - 3s 35ms/step - loss: 1.5637 - accuracy: 0.43
29 - val_loss: 1.5571 - val_accuracy: 0.4396
Epoch 3/20
75/75 [==============================] - 3s 37ms/step - loss: 1.4284 - accuracy: 0.48
12 - val_loss: 1.5118 - val_accuracy: 0.4612
Epoch 4/20
75/75 [==============================] - 3s 40ms/step - loss: 1.3741 - accuracy: 0.50
01 - val_loss: 1.4519 - val_accuracy: 0.4716
Epoch 5/20
75/75 [==============================] - 3s 38ms/step - loss: 1.2785 - accuracy: 0.53
63 - val_loss: 1.4655 - val_accuracy: 0.4736
Epoch 6/20
75/75 [==============================] - 3s 36ms/step - loss: 1.2505 - accuracy: 0.54
08 - val_loss: 1.5603 - val_accuracy: 0.4712
Epoch 7/20
75/75 [==============================] - 2s 32ms/step - loss: 1.1663 - accuracy: 0.56
84 - val_loss: 1.4924 - val_accuracy: 0.4712
Epoch 8/20
75/75 [==============================] - 3s 38ms/step - loss: 1.1180 - accuracy: 0.59
21 - val_loss: 1.4943 - val_accuracy: 0.4880
Epoch 9/20
75/75 [==============================] - 3s 39ms/step - loss: 1.0609 - accuracy: 0.60
89 - val_loss: 1.5651 - val_accuracy: 0.4796
Epoch 10/20
75/75 [==============================] - 3s 38ms/step - loss: 0.9905 - accuracy: 0.63
51 - val_loss: 1.5768 - val_accuracy: 0.4748
Epoch 11/20
75/75 [==============================] - 3s 39ms/step - loss: 0.9283 - accuracy: 0.65
68 - val_loss: 1.6821 - val_accuracy: 0.4684
Epoch 12/20
75/75 [==============================] - 3s 39ms/step - loss: 0.9051 - accuracy: 0.66
63 - val_loss: 1.7326 - val_accuracy: 0.4680
Epoch 13/20
75/75 [==============================] - 3s 37ms/step - loss: 0.8427 - accuracy: 0.69
32 - val_loss: 1.8039 - val_accuracy: 0.4720
Epoch 14/20
75/75 [==============================] - 3s 38ms/step - loss: 0.7878 - accuracy: 0.70
91 - val_loss: 1.8690 - val_accuracy: 0.4716
Epoch 15/20
75/75 [==============================] - 3s 45ms/step - loss: 0.7598 - accuracy: 0.72
35 - val_loss: 1.8614 - val_accuracy: 0.4580
Epoch 16/20
75/75 [==============================] - 3s 41ms/step - loss: 0.6991 - accuracy: 0.74
63 - val_loss: 2.0554 - val_accuracy: 0.4700
Epoch 17/20
75/75 [==============================] - 3s 39ms/step - loss: 0.6503 - accuracy: 0.76
45 - val_loss: 2.1389 - val_accuracy: 0.4616
Epoch 18/20
75/75 [==============================] - 3s 38ms/step - loss: 0.6607 - accuracy: 0.76
53 - val_loss: 2.1597 - val_accuracy: 0.4688
Epoch 19/20
75/75 [==============================] - 3s 39ms/step - loss: 0.5708 - accuracy: 0.78
97 - val_loss: 2.5446 - val_accuracy: 0.4604
Epoch 20/20
75/75 [==============================] - 3s 40ms/step - loss: 0.5661 - accuracy: 0.79
12 - val_loss: 2.5285 - val_accuracy: 0.4512
```

In [44]:
```
# Evaluate the trained model on test set, not used in training or validation
score = model2.evaluate(Xtest, Ytest, verbose=1)
print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])
```

```
63/63 [==============================] - 2s 21ms/step - loss: 2.5789 - accuracy: 0.45
10
Test loss: 2.5789
Test accuracy: 0.4510
```

In [45]:
```
# Plot the history from the training run
plot_results(history2)
```





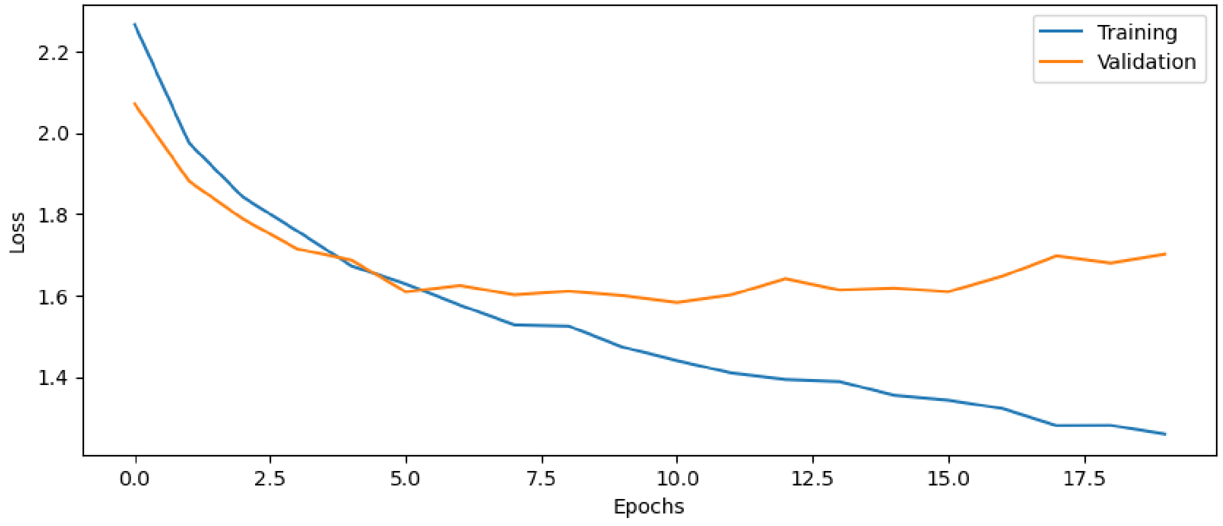
4 convolutional layers, 1 intermediate dense layer (50 nodes)

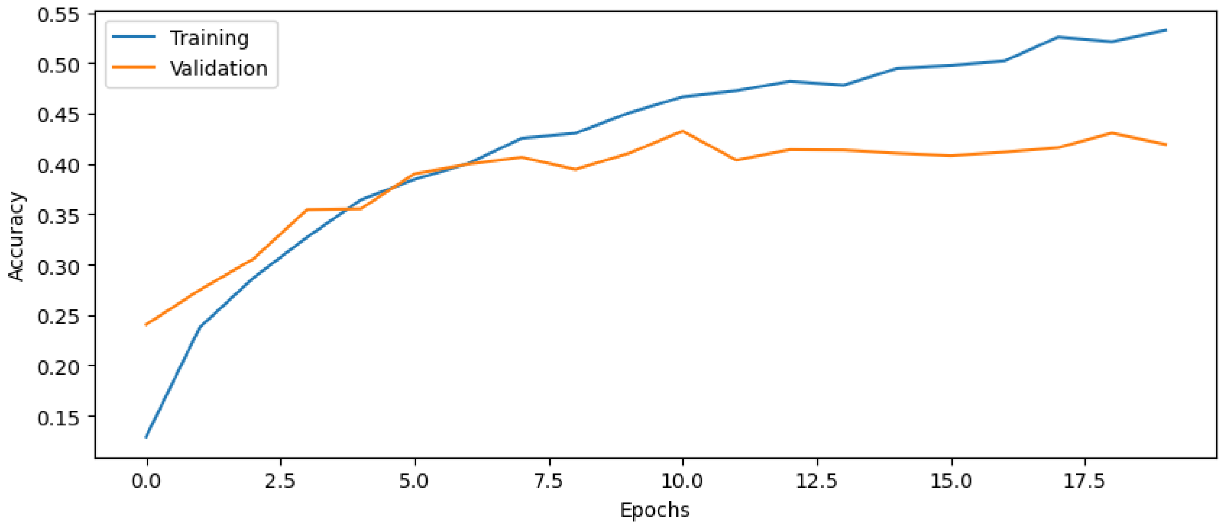
In [46]:
```
# Setup some training parameters
batch_size = 100
epochs = 20
input_shape = Xtrain.shape[1:4]

# Build model
model3 = build_CNN(input_shape, n_conv_layers=4, n_filters=16, n_dense_layers=1, n_noc
# Train the model  using training data and validation data
history3 = model3.fit(Xtrain,Ytrain, batch_size=batch_size,epochs=epochs, verbose=1, v
Epoch 1/20
75/75 [==============================] - 13s 100ms/step - loss: 2.2659 - accuracy: 0.
1291 - val_loss: 2.0710 - val_accuracy: 0.2404
Epoch 2/20
75/75 [==============================] - 4s 56ms/step - loss: 1.9752 - accuracy: 0.23
76 - val_loss: 1.8814 - val_accuracy: 0.2748
Epoch 3/20
75/75 [==============================] - 5s 68ms/step - loss: 1.8416 - accuracy: 0.28
68 - val_loss: 1.7880 - val_accuracy: 0.3056
Epoch 4/20
75/75 [==============================] - 4s 59ms/step - loss: 1.7586 - accuracy: 0.32
71 - val_loss: 1.7147 - val_accuracy: 0.3544
Epoch 5/20
75/75 [==============================] - 4s 59ms/step - loss: 1.6730 - accuracy: 0.36
41 - val_loss: 1.6871 - val_accuracy: 0.3552
Epoch 6/20
75/75 [==============================] - 5s 65ms/step - loss: 1.6287 - accuracy: 0.38
44 - val_loss: 1.6093 - val_accuracy: 0.3900
Epoch 7/20
75/75 [==============================] - 5s 67ms/step - loss: 1.5767 - accuracy: 0.40
01 - val_loss: 1.6250 - val_accuracy: 0.3996
Epoch 8/20
75/75 [==============================] - 5s 65ms/step - loss: 1.5285 - accuracy: 0.42
52 - val_loss: 1.6018 - val_accuracy: 0.4064
Epoch 9/20
75/75 [==============================] - 5s 61ms/step - loss: 1.5253 - accuracy: 0.43
03 - val_loss: 1.6110 - val_accuracy: 0.3944
Epoch 10/20
75/75 [==============================] - 5s 65ms/step - loss: 1.4736 - accuracy: 0.45
04 - val_loss: 1.6000 - val_accuracy: 0.4104
Epoch 11/20
75/75 [==============================] - 5s 62ms/step - loss: 1.4404 - accuracy: 0.46
64 - val_loss: 1.5834 - val_accuracy: 0.4324
Epoch 12/20
75/75 [==============================] - 5s 62ms/step - loss: 1.4099 - accuracy: 0.47
24 - val_loss: 1.6019 - val_accuracy: 0.4036
Epoch 13/20
75/75 [==============================] - 5s 61ms/step - loss: 1.3941 - accuracy: 0.48
17 - val_loss: 1.6415 - val_accuracy: 0.4140
Epoch 14/20
75/75 [==============================] - 5s 65ms/step - loss: 1.3889 - accuracy: 0.47
79 - val_loss: 1.6138 - val_accuracy: 0.4136
Epoch 15/20
75/75 [==============================] - 5s 64ms/step - loss: 1.3551 - accuracy: 0.49
47 - val_loss: 1.6186 - val_accuracy: 0.4104
Epoch 16/20
75/75 [==============================] - 4s 58ms/step - loss: 1.3432 - accuracy: 0.49
75 - val_loss: 1.6095 - val_accuracy: 0.4080
Epoch 17/20
75/75 [==============================] - 4s 54ms/step - loss: 1.3226 - accuracy: 0.50
20 - val_loss: 1.6476 - val_accuracy: 0.4116
Epoch 18/20
75/75 [==============================] - 4s 56ms/step - loss: 1.2809 - accuracy: 0.52
57 - val_loss: 1.6975 - val_accuracy: 0.4160
Epoch 19/20
75/75 [==============================] - 4s 54ms/step - loss: 1.2813 - accuracy: 0.52
11 - val_loss: 1.6801 - val_accuracy: 0.4304
Epoch 20/20
75/75 [==============================] - 5s 68ms/step - loss: 1.2597 - accuracy: 0.53
24 - val_loss: 1.7016 - val_accuracy: 0.4192
```

In [48]:
```python
# Evaluate the trained model on test set, not used in training or validation
score = model3.evaluate(Xtest,Ytest,verbose=1)
print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])
```

```
63/63 [==============================] - 3s 22ms/step - loss: 1.6751 - accuracy: 0.42
40
Test loss: 1.6751
Test accuracy: 0.4240
```

In [49]:
```python
# Plot the history from the training run
plot_results(history3)
```





Part 13: Plot the CNN architecture

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

Question 12: How many trainable parameters does your network have? Which part of the network contains most of the parameters?

Answer:

Network has 123,600 trainable parameters. conv2d_11 has most number of parameters. i.e 73856

Question 13: What is the input to and output of a Conv2D layer? What are the dimensions of the input and output?

Answer:

The input to a Conv2D layer is tensor that has dimendions - batch_size, heigh of images, width of images, channels

The output of a Conv2D layer is another 4D tensor with a new shape: (batch_size, new_height, new_width, filters), where:

- batch_size : the same as the input
- new_height : the height of the output feature maps
- new_width : the width of the output feature maps
- filters : the number of filters in the Conv2D layer

Input dimensions : (100 , 32 , 32, 3)
Output dimensions : (None, 32, 32, 16)

Question 14: Is the batch size always the first dimension of each 4D tensor? Check the documentation for Conv2D, https://keras.io/layers/convolutional/

Answer:

Yes, the batch size is always the first dimension of each 4D tensor.

Question 15: 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?

Answer:

The number of channels in the output will also be 128, because each filter produces a single channel in the output, and the number of filters in the convolutional layer does not affect the number of channels in the input or output.

Question 16: 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)?

Answer:

The number of parameters in a Conv2D layer is calculated as (shape of filter x number of filters in the previous layer (depth) + 1) x (number of filters). This formula takes into account the number of filter coefficients, the number of filters, and the number of biases.

Question 17: How does MaxPooling help in reducing the number of parameters to train?

Answer:

MaxPooling helps in reducing the number of parameters to train by downsampling the output feature maps from the Conv2D layer. This operation selects the maximum value from a region of the feature map, and this reduces the size of the feature map while retaining the most important information.

In [50]:
```
# Print network architecture

model3.summary()
```

```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 conv2d_8 (Conv2D)           (None, 32, 32, 16)        448       
                                                                 
 max_pooling2d_8 (MaxPooling  (None, 16, 16, 16)       0         
 2D)                                                             
                                                                 
 conv2d_9 (Conv2D)           (None, 16, 16, 32)        4640      
                                                                 
 max_pooling2d_9 (MaxPooling  (None, 8, 8, 32)         0         
 2D)                                                             
                                                                 
 conv2d_10 (Conv2D)          (None, 8, 8, 64)          18496     
                                                                 
 max_pooling2d_10 (MaxPoolin  (None, 4, 4, 64)         0         
 g2D)                                                            
                                                                 
 conv2d_11 (Conv2D)          (None, 4, 4, 128)         73856     
                                                                 
 max_pooling2d_11 (MaxPoolin  (None, 2, 2, 128)        0         
 g2D)                                                            
                                                                 
 flatten_4 (Flatten)         (None, 512)               0         
                                                                 
 dense_5 (Dense)             (None, 50)                25650     
                                                                 
 dense_6 (Dense)             (None, 10)                510       
                                                                 
=================================================================
Total params: 123,600
Trainable params: 123,600
Non-trainable params: 0
_________________________________________________________________
```

Part 14: Dropout regularization

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

Question 18: How much did the test accuracy improve with dropout, compared to without dropout?

Answer:

Accuracy is reduced to 37.10% from 42.40%

Question 19: What other types of regularization can be applied? How can you add L2 regularization for the convolutional layers?

Answer:

In addition to dropout, we can use other types of regularization techniques like L1 regularization and L2 regularization. L1 regularization adds a penalty term to the loss function proportional to the absolute value of the weights which makes the model to use fewer features in the input and can help with feature selection L2 regularization adds a penalty term to the loss function proportional to the squared value of the weights which makes the model to use smaller weights and can help prevent overfitting.
To add L2 regularization for the convolutional layers in Keras, we can use the kernel_regularizer argument when defining the layers.

4 convolutional layers, 1 intermediate dense layer (50 nodes), dropout

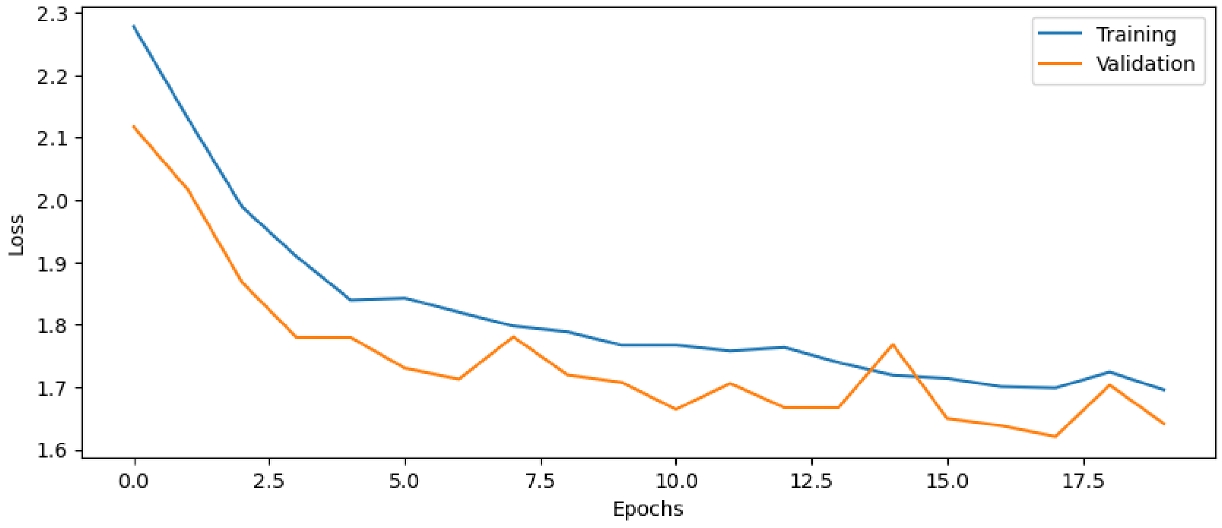
In [51]:

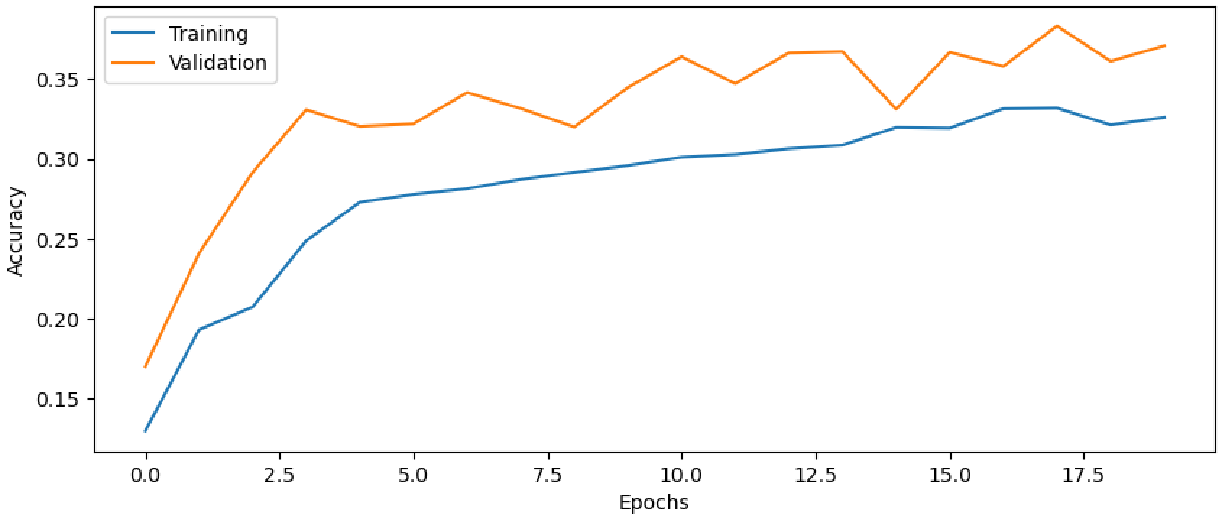
```
# Setup some training parameters
batch_size = 100
epochs = 20
input_shape = Xtrain.shape[1:4]

# Build model
model4 = build_CNN(input_shape, n_conv_layers=4, n_filters=16, n_dense_layers=1, n_noc

# Train the model  using training data and validation data
history4 = model4.fit(Xtrain,Ytrain, batch_size=batch_size,epochs=epochs, verbose=1, v
Epoch 1/20
75/75 [==============================] - 12s 105ms/step - loss: 2.2777 - accuracy: 0.
1297 - val_loss: 2.1172 - val_accuracy: 0.1700
Epoch 2/20
75/75 [==============================] - 5s 60ms/step - loss: 2.1303 - accuracy: 0.19
31 - val_loss: 2.0162 - val_accuracy: 0.2408
Epoch 3/20
75/75 [==============================] - 5s 62ms/step - loss: 1.9888 - accuracy: 0.20
73 - val_loss: 1.8677 - val_accuracy: 0.2916
Epoch 4/20
75/75 [==============================] - 5s 61ms/step - loss: 1.9095 - accuracy: 0.24
88 - val_loss: 1.7800 - val_accuracy: 0.3308
Epoch 5/20
75/75 [==============================] - 4s 57ms/step - loss: 1.8397 - accuracy: 0.27
29 - val_loss: 1.7799 - val_accuracy: 0.3204
Epoch 6/20
75/75 [==============================] - 5s 61ms/step - loss: 1.8429 - accuracy: 0.27
79 - val_loss: 1.7305 - val_accuracy: 0.3220
Epoch 7/20
75/75 [==============================] - 5s 67ms/step - loss: 1.8201 - accuracy: 0.28
15 - val_loss: 1.7130 - val_accuracy: 0.3416
Epoch 8/20
75/75 [==============================] - 5s 65ms/step - loss: 1.7981 - accuracy: 0.28
72 - val_loss: 1.7808 - val_accuracy: 0.3316
Epoch 9/20
75/75 [==============================] - 5s 62ms/step - loss: 1.7887 - accuracy: 0.29
16 - val_loss: 1.7197 - val_accuracy: 0.3200
Epoch 10/20
75/75 [==============================] - 5s 63ms/step - loss: 1.7675 - accuracy: 0.29
59 - val_loss: 1.7077 - val_accuracy: 0.3448
Epoch 11/20
75/75 [==============================] - 5s 62ms/step - loss: 1.7675 - accuracy: 0.30
11 - val_loss: 1.6647 - val_accuracy: 0.3640
Epoch 12/20
75/75 [==============================] - 5s 64ms/step - loss: 1.7581 - accuracy: 0.30
28 - val_loss: 1.7059 - val_accuracy: 0.3472
Epoch 13/20
75/75 [==============================] - 5s 62ms/step - loss: 1.7638 - accuracy: 0.30
64 - val_loss: 1.6674 - val_accuracy: 0.3664
Epoch 14/20
75/75 [==============================] - 5s 64ms/step - loss: 1.7395 - accuracy: 0.30
87 - val_loss: 1.6674 - val_accuracy: 0.3672
Epoch 15/20
75/75 [==============================] - 5s 65ms/step - loss: 1.7193 - accuracy: 0.31
97 - val_loss: 1.7685 - val_accuracy: 0.3312
Epoch 16/20
75/75 [==============================] - 5s 63ms/step - loss: 1.7142 - accuracy: 0.31
93 - val_loss: 1.6500 - val_accuracy: 0.3668
Epoch 17/20
75/75 [==============================] - 5s 67ms/step - loss: 1.7008 - accuracy: 0.33
15 - val_loss: 1.6383 - val_accuracy: 0.3580
Epoch 18/20
75/75 [==============================] - 5s 64ms/step - loss: 1.6988 - accuracy: 0.33
19 - val_loss: 1.6210 - val_accuracy: 0.3832
Epoch 19/20
75/75 [==============================] - 5s 63ms/step - loss: 1.7244 - accuracy: 0.32
13 - val_loss: 1.7034 - val_accuracy: 0.3612
Epoch 20/20
75/75 [==============================] - 5s 61ms/step - loss: 1.6955 - accuracy: 0.32
60 - val_loss: 1.6415 - val_accuracy: 0.3708
```

In [52]:
```
# Evaluate the trained model on test set, not used in training or validation
score = model4.evaluate(Xtest,Ytest,verbose=1)
print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])
```

```
63/63 [==============================] - 3s 28ms/step - loss: 1.6108 - accuracy: 0.37
10
Test loss: 1.6108
Test accuracy: 0.3710
```

In [53]:
```
# Plot the history from the training run
plot_results(history4)
```





Part 15: Tweaking 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.

Question 20: How high test accuracy can you obtain? What is your best configuration?

Answer:

The highest test accuracy obtained is 49.80% with the following configuration

- convolutional layers : 2
- number of filters per layer : 12
- number of intermediate dense layers : 2
- number of nodes in the intermediate dense layers : 20
- batch size : 100
- learning rate : 0.01
- number of epochs: 20

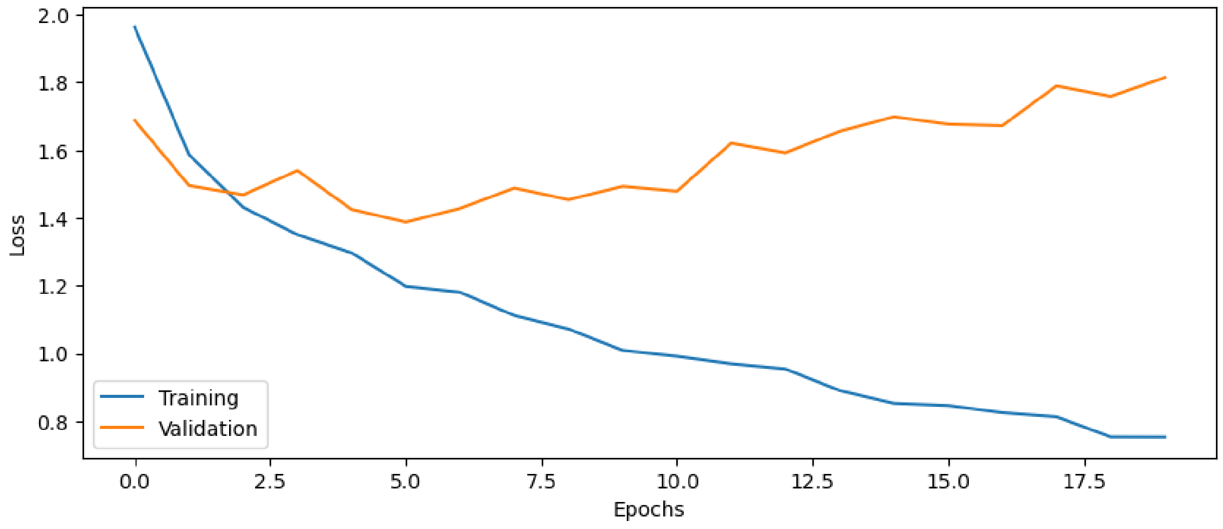
Your best config

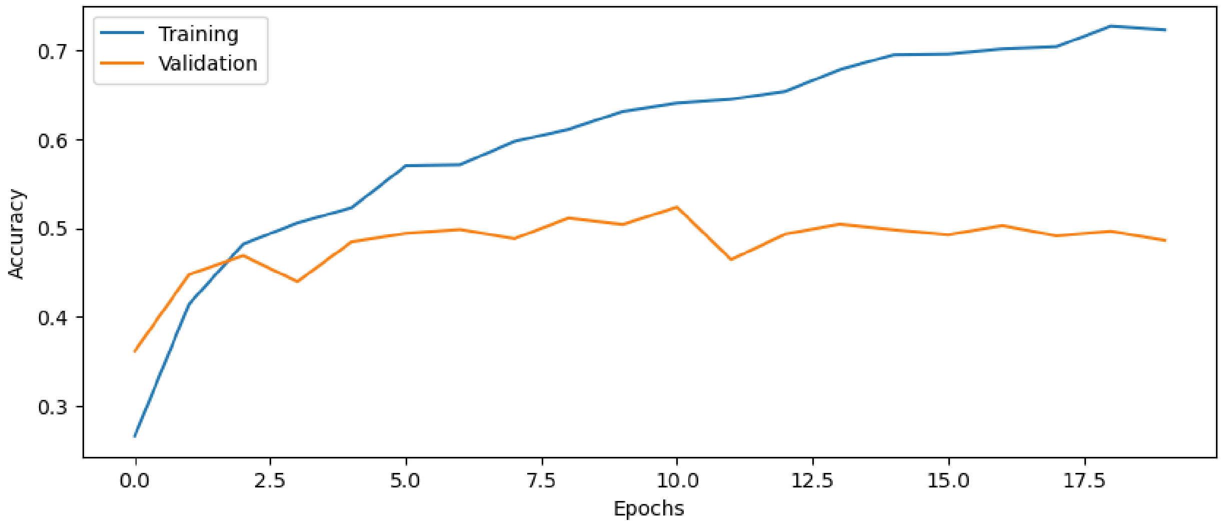
In [73]:
```
# Setup some training parameters
epochs=20
# Build model
model5 = build_CNN(input_shape, n_conv_layers=2, n_filters=12, n_dense_layers=2, n_noc

# Train the model  using training data and validation data
history5 = model5.fit(Xtrain,Ytrain, batch_size=batch_size,epochs=epochs, verbose=1, v
Epoch 1/20
75/75 [==============================] - 9s 78ms/step - loss: 1.9625 - accuracy: 0.26
55 - val_loss: 1.6873 - val_accuracy: 0.3612
Epoch 2/20
75/75 [==============================] - 3s 37ms/step - loss: 1.5861 - accuracy: 0.41
41 - val_loss: 1.4952 - val_accuracy: 0.4476
Epoch 3/20
75/75 [==============================] - 3s 40ms/step - loss: 1.4308 - accuracy: 0.48
19 - val_loss: 1.4676 - val_accuracy: 0.4688
Epoch 4/20
75/75 [==============================] - 3s 41ms/step - loss: 1.3501 - accuracy: 0.50
57 - val_loss: 1.5393 - val_accuracy: 0.4396
Epoch 5/20
75/75 [==============================] - 3s 36ms/step - loss: 1.2962 - accuracy: 0.52
29 - val_loss: 1.4245 - val_accuracy: 0.4844
Epoch 6/20
75/75 [==============================] - 3s 39ms/step - loss: 1.1977 - accuracy: 0.57
03 - val_loss: 1.3872 - val_accuracy: 0.4940
Epoch 7/20
75/75 [==============================] - 3s 36ms/step - loss: 1.1801 - accuracy: 0.57
13 - val_loss: 1.4276 - val_accuracy: 0.4984
Epoch 8/20
75/75 [==============================] - 3s 36ms/step - loss: 1.1115 - accuracy: 0.59
76 - val_loss: 1.4878 - val_accuracy: 0.4880
Epoch 9/20
75/75 [==============================] - 3s 36ms/step - loss: 1.0715 - accuracy: 0.61
13 - val_loss: 1.4534 - val_accuracy: 0.5112
Epoch 10/20
75/75 [==============================] - 3s 37ms/step - loss: 1.0086 - accuracy: 0.63
15 - val_loss: 1.4928 - val_accuracy: 0.5040
Epoch 11/20
75/75 [==============================] - 3s 36ms/step - loss: 0.9912 - accuracy: 0.64
08 - val_loss: 1.4784 - val_accuracy: 0.5236
Epoch 12/20
75/75 [==============================] - 3s 38ms/step - loss: 0.9691 - accuracy: 0.64
52 - val_loss: 1.6202 - val_accuracy: 0.4644
Epoch 13/20
75/75 [==============================] - 3s 36ms/step - loss: 0.9536 - accuracy: 0.65
41 - val_loss: 1.5917 - val_accuracy: 0.4932
Epoch 14/20
75/75 [==============================] - 3s 36ms/step - loss: 0.8900 - accuracy: 0.67
85 - val_loss: 1.6553 - val_accuracy: 0.5044
Epoch 15/20
75/75 [==============================] - 3s 39ms/step - loss: 0.8522 - accuracy: 0.69
55 - val_loss: 1.6980 - val_accuracy: 0.4980
Epoch 16/20
75/75 [==============================] - 3s 40ms/step - loss: 0.8457 - accuracy: 0.69
63 - val_loss: 1.6764 - val_accuracy: 0.4924
Epoch 17/20
75/75 [==============================] - 3s 42ms/step - loss: 0.8248 - accuracy: 0.70
20 - val_loss: 1.6720 - val_accuracy: 0.5028
Epoch 18/20
75/75 [==============================] - 3s 39ms/step - loss: 0.8125 - accuracy: 0.70
45 - val_loss: 1.7892 - val_accuracy: 0.4912
Epoch 19/20
75/75 [==============================] - 3s 39ms/step - loss: 0.7532 - accuracy: 0.72
76 - val_loss: 1.7583 - val_accuracy: 0.4964
Epoch 20/20
75/75 [==============================] - 3s 39ms/step - loss: 0.7528 - accuracy: 0.72
35 - val_loss: 1.8140 - val_accuracy: 0.4860
```

In [74]:
```python
# Evaluate the trained model on test set, not used in training or validation
score = model5.evaluate(Xtest,Ytest)
print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])
```

```
63/63 [==============================] - 3s 21ms/step - loss: 1.7227 - accuracy: 0.49
80
Test loss: 1.7227
Test accuracy: 0.4980
```

In [75]:
```python
# Plot the history from the training run
plot_results(history5)
```





Part 16: Rotate the test images

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.

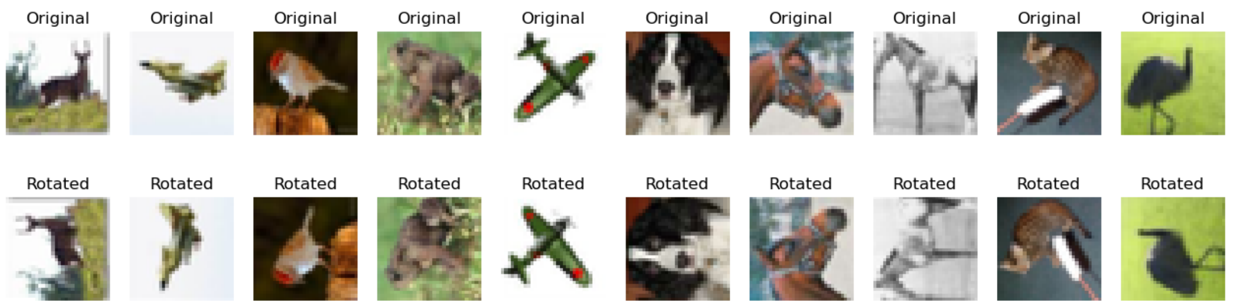
Question 21: What is the test accuracy for rotated test images, compared to test images without rotation? Explain the difference in accuracy.

Answer:

The test accuracy for rotated test images is 23.05% which is far lesser than the accuracy of test images without rotation. The reason for the difference in accuracy is the model was trained with non-rotated images and tested on rotated images.

In [76]:
```python
def myrotate(images):

    images_rot = np.rot90(images, axes=(1,2))

    return images_rot
```

In [77]:
```python
# 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()
```



In [78]:
```python
# Evaluate the trained model on rotated test set
score = model5.evaluate(Xtest_rotated,Ytest)
print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])
```

```
63/63 [==============================] - 2s 24ms/step - loss: 3.9957 - accuracy: 0.23
05
Test loss: 3.9957
Test accuracy: 0.2305
```

Part 17: 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`

See https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator , the .flow(x,y) functionality

Make sure to use different subsets for training and validation when you setup the flows, otherwise you will validate on the same data...

In [83]:
```
# Get all 60 000 training images again. ImageDataGenerator manages validation data on
(Xtrain, Ytrain), _ = cifar10.load_data()

# Reduce number of images to 10,000
Xtrain = Xtrain[0:10000]
Ytrain = Ytrain[0:10000]

# Change data type and rescale range
Xtrain = Xtrain.astype('float32')
Xtrain = Xtrain / 127.5 - 1

# Convert labels to hot encoding
Ytrain = to_categorical(Ytrain, 10)
```

In [84]:
```
# Set up a data generator with on-the-fly data augmentation, 20% validation split
Xtrain, Xval, Ytrain, Yval = train_test_split(Xtrain, Ytrain,test_size=0.20, random_st

# Use a rotation range of 30 degrees, horizontal and vertical flipping
from keras.preprocessing.image import ImageDataGenerator
datagen = ImageDataGenerator(rotation_range=30, horizontal_flip=True,vertical_flip=Tru


# Setup a flow for training data, assume that we can fit all images into CPU memory
train_data = datagen.flow(Xtrain, Ytrain, batch_size=batch_size)


# Setup a flow for validation data, assume that we can fit all images into CPU memory
valid_data = datagen.flow(Xval, Yval, batch_size=batch_size)
```

Part 18: What about big data?

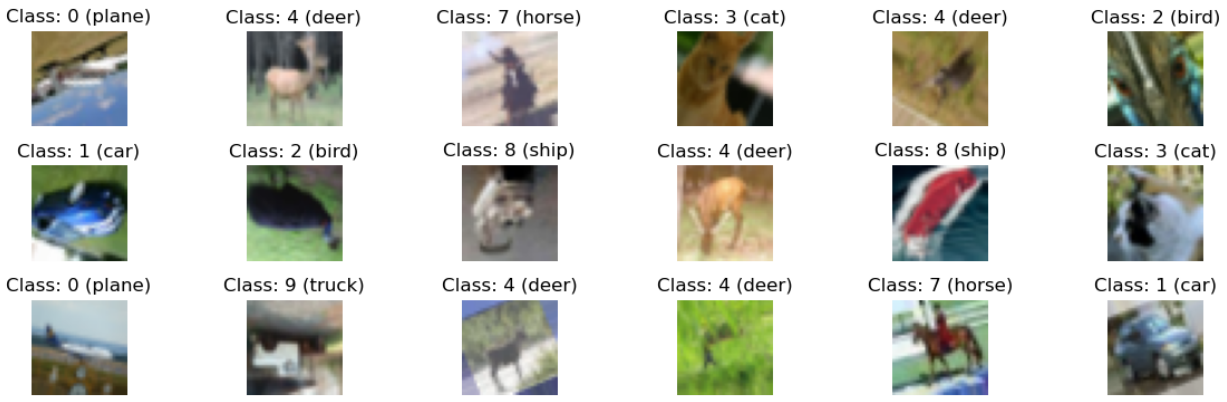
Question 22: How would you change the code for the image generator if you cannot fit all training images in CPU memory? What is the disadvantage of doing that change?

Answer:

If we cannot fit all training images in CPU memory, we can use the flow_from_directory method of ImageDataGenerator instead of flow method. This method reads the images from disk in batches and applies data augmentation on the fly.

The disadvantage of using this method is that reading images from disk can be slower than loading images into memory which can result in slower training durations.

In [85]:
```
# Plot some augmented images
plot_datagen = datagen.flow(Xtrain, Ytrain, batch_size=1)

plt.figure(figsize=(12,4))
for i in range(18):
    (im, label) = plot_datagen.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()
```



Part 19: Train the CNN with images from the generator

See https://keras.io/api/models/model_training_apis/#fit-method for how to use model.fit 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)*(1 -
validation_split)/batch_size

validation_steps should be set to:
len(Xtrain)*validation_split/batch_size
```

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

Question 23: 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?

Answer:

Training accuracy is increasing faster compared to without augmentation. This is because data augmentation creates additional training data by applying transformation(rotation) to the existing samples, which helps the model to generalize better and learn more robust features.

Question 24: What other types of image augmentation can be applied, compared to what we use here?

Answer:

Other types of image augmentation are

- Flipping: mirroring the image along the vertical or horizontal axis.
- Scaling: zooming in or out of the image by a small amount.
- Cropping: randomly crops a section of the image and resize it back to the original size.
- Color Jittering: changing the color intensity, brightness, contrast, and saturation of the image.
- Noise: Adding random noise to the image

In [95]:
```
# Setup some training parameters
batch_size = 100
epochs = 200
input_shape = Xtrain.shape[1:4]

# Build model (your best config)
model6 = build_CNN(input_shape, n_conv_layers=2, n_filters=12,n_dense_layers=2, n_node

train_datagen = datagen.flow(Xtrain, Ytrain, batch_size=batch_size)
valid_datagen = datagen.flow(Xtrain, Ytrain, batch_size=batch_size)
validation_split=0.2


# Train the model using on the fly augmentation
history6 = model6.fit_generator(train_datagen,steps_per_epoch=len(Xtrain)*(1 - validat
```

```
C:\Users\Dell\AppData\Local\Temp\ipykernel_15876\3141439534.py:15: UserWarning: `Mode
l.fit_generator` is deprecated and will be removed in a future version. Please use `M
odel.fit`, which supports generators.
  history6 = model6.fit_generator(train_datagen,steps_per_epoch=len(Xtrain)*(1 - vali
dation_split)/batch_size, epochs=epochs, validation_data=valid_datagen,validation_ste
ps=len(Xtrain)*validation_split/batch_size)
```

```
Epoch 1/200
64/64 [==============================] - 18s 218ms/step - loss: 2.1098 - accuracy: 0.
2028 - val_loss: 1.9443 - val_accuracy: 0.2488
Epoch 2/200
64/64 [==============================] - 11s 165ms/step - loss: 1.9191 - accuracy: 0.
2697 - val_loss: 1.8553 - val_accuracy: 0.2969
Epoch 3/200
64/64 [==============================] - 11s 172ms/step - loss: 1.8191 - accuracy: 0.
3092 - val_loss: 1.7538 - val_accuracy: 0.2950
Epoch 4/200
64/64 [==============================] - 11s 175ms/step - loss: 1.7975 - accuracy: 0.
3206 - val_loss: 1.7230 - val_accuracy: 0.3650
Epoch 5/200
64/64 [==============================] - 11s 176ms/step - loss: 1.7159 - accuracy: 0.
3500 - val_loss: 1.6779 - val_accuracy: 0.3762
Epoch 6/200
64/64 [==============================] - 11s 177ms/step - loss: 1.6728 - accuracy: 0.
3762 - val_loss: 1.7054 - val_accuracy: 0.3775
Epoch 7/200
64/64 [==============================] - 11s 178ms/step - loss: 1.6577 - accuracy: 0.
3789 - val_loss: 1.5960 - val_accuracy: 0.4119
Epoch 8/200
64/64 [==============================] - 11s 166ms/step - loss: 1.6404 - accuracy: 0.
3848 - val_loss: 1.5996 - val_accuracy: 0.4013
Epoch 9/200
64/64 [==============================] - 11s 169ms/step - loss: 1.6073 - accuracy: 0.
3956 - val_loss: 1.5560 - val_accuracy: 0.4256
Epoch 10/200
64/64 [==============================] - 11s 171ms/step - loss: 1.5958 - accuracy: 0.
3988 - val_loss: 1.6027 - val_accuracy: 0.4112
Epoch 11/200
64/64 [==============================] - 11s 179ms/step - loss: 1.6121 - accuracy: 0.
3980 - val_loss: 1.5591 - val_accuracy: 0.4094
Epoch 12/200
64/64 [==============================] - 10s 162ms/step - loss: 1.5781 - accuracy: 0.
4144 - val_loss: 1.5174 - val_accuracy: 0.4406
Epoch 13/200
64/64 [==============================] - 10s 163ms/step - loss: 1.5738 - accuracy: 0.
4255 - val_loss: 1.5451 - val_accuracy: 0.4338
Epoch 14/200
64/64 [==============================] - 11s 168ms/step - loss: 1.5646 - accuracy: 0.
4228 - val_loss: 1.4515 - val_accuracy: 0.4544
Epoch 15/200
64/64 [==============================] - 11s 176ms/step - loss: 1.5399 - accuracy: 0.
4317 - val_loss: 1.5347 - val_accuracy: 0.4338
Epoch 16/200
64/64 [==============================] - 11s 176ms/step - loss: 1.5310 - accuracy: 0.
4437 - val_loss: 1.5390 - val_accuracy: 0.4481
Epoch 17/200
64/64 [==============================] - 11s 172ms/step - loss: 1.5600 - accuracy: 0.
4223 - val_loss: 1.5987 - val_accuracy: 0.4038
Epoch 18/200
64/64 [==============================] - 11s 176ms/step - loss: 1.5303 - accuracy: 0.
4394 - val_loss: 1.5271 - val_accuracy: 0.4387
Epoch 19/200
64/64 [==============================] - 11s 174ms/step - loss: 1.5224 - accuracy: 0.
4445 - val_loss: 1.5392 - val_accuracy: 0.4494
Epoch 20/200
64/64 [==============================] - 11s 169ms/step - loss: 1.5344 - accuracy: 0.
4467 - val_loss: 1.5757 - val_accuracy: 0.4256
```

```
Epoch 21/200
64/64 [==============================] - 10s 163ms/step - loss: 1.5103 - accuracy: 0.
4397 - val_loss: 1.5598 - val_accuracy: 0.4162
Epoch 22/200
64/64 [==============================] - 11s 166ms/step - loss: 1.5194 - accuracy: 0.
4467 - val_loss: 1.5005 - val_accuracy: 0.4538
Epoch 23/200
64/64 [==============================] - 11s 171ms/step - loss: 1.4952 - accuracy: 0.
4459 - val_loss: 1.5198 - val_accuracy: 0.4356
Epoch 24/200
64/64 [==============================] - 12s 184ms/step - loss: 1.5101 - accuracy: 0.
4408 - val_loss: 1.4689 - val_accuracy: 0.4675
Epoch 25/200
64/64 [==============================] - 11s 178ms/step - loss: 1.4872 - accuracy: 0.
4480 - val_loss: 1.5122 - val_accuracy: 0.4319
Epoch 26/200
64/64 [==============================] - 11s 175ms/step - loss: 1.4831 - accuracy: 0.
4508 - val_loss: 1.4600 - val_accuracy: 0.4538
Epoch 27/200
64/64 [==============================] - 11s 171ms/step - loss: 1.4912 - accuracy: 0.
4564 - val_loss: 1.4679 - val_accuracy: 0.4619
Epoch 28/200
64/64 [==============================] - 11s 168ms/step - loss: 1.4635 - accuracy: 0.
4650 - val_loss: 1.4324 - val_accuracy: 0.4775
Epoch 29/200
64/64 [==============================] - 11s 169ms/step - loss: 1.4646 - accuracy: 0.
4739 - val_loss: 1.4183 - val_accuracy: 0.4600
Epoch 30/200
64/64 [==============================] - 11s 174ms/step - loss: 1.4825 - accuracy: 0.
4558 - val_loss: 1.4665 - val_accuracy: 0.4531
Epoch 31/200
64/64 [==============================] - 11s 165ms/step - loss: 1.4844 - accuracy: 0.
4544 - val_loss: 1.4309 - val_accuracy: 0.4644
Epoch 32/200
64/64 [==============================] - 11s 165ms/step - loss: 1.4905 - accuracy: 0.
4569 - val_loss: 1.4486 - val_accuracy: 0.4700
Epoch 33/200
64/64 [==============================] - 11s 169ms/step - loss: 1.4499 - accuracy: 0.
4694 - val_loss: 1.4596 - val_accuracy: 0.4737
Epoch 34/200
64/64 [==============================] - 11s 174ms/step - loss: 1.4614 - accuracy: 0.
4675 - val_loss: 1.5364 - val_accuracy: 0.4450
Epoch 35/200
64/64 [==============================] - 11s 171ms/step - loss: 1.4575 - accuracy: 0.
4636 - val_loss: 1.4616 - val_accuracy: 0.4681
Epoch 36/200
64/64 [==============================] - 11s 168ms/step - loss: 1.4457 - accuracy: 0.
4709 - val_loss: 1.3651 - val_accuracy: 0.4950
Epoch 37/200
64/64 [==============================] - 11s 167ms/step - loss: 1.4688 - accuracy: 0.
4663 - val_loss: 1.4383 - val_accuracy: 0.4812
Epoch 38/200
64/64 [==============================] - 11s 174ms/step - loss: 1.4557 - accuracy: 0.
4730 - val_loss: 1.4066 - val_accuracy: 0.4750
Epoch 39/200
64/64 [==============================] - 11s 176ms/step - loss: 1.4513 - accuracy: 0.
4752 - val_loss: 1.3761 - val_accuracy: 0.4919
Epoch 40/200
64/64 [==============================] - 11s 174ms/step - loss: 1.4314 - accuracy: 0.
4742 - val_loss: 1.4070 - val_accuracy: 0.4831
```

```
Epoch 41/200
64/64 [==============================] - 11s 173ms/step - loss: 1.4486 - accuracy: 0.
4712 - val_loss: 1.4285 - val_accuracy: 0.4863
Epoch 42/200
64/64 [==============================] - 11s 175ms/step - loss: 1.4449 - accuracy: 0.
4791 - val_loss: 1.4740 - val_accuracy: 0.4706
Epoch 43/200
64/64 [==============================] - 11s 179ms/step - loss: 1.4546 - accuracy: 0.
4770 - val_loss: 1.4451 - val_accuracy: 0.4725
Epoch 44/200
64/64 [==============================] - 11s 172ms/step - loss: 1.4485 - accuracy: 0.
4737 - val_loss: 1.3795 - val_accuracy: 0.4938
Epoch 45/200
64/64 [==============================] - 11s 171ms/step - loss: 1.4459 - accuracy: 0.
4734 - val_loss: 1.4501 - val_accuracy: 0.4819
Epoch 46/200
64/64 [==============================] - 11s 168ms/step - loss: 1.4487 - accuracy: 0.
4716 - val_loss: 1.4342 - val_accuracy: 0.4675
Epoch 47/200
64/64 [==============================] - 11s 179ms/step - loss: 1.4166 - accuracy: 0.
4828 - val_loss: 1.4000 - val_accuracy: 0.4913
Epoch 48/200
64/64 [==============================] - 12s 185ms/step - loss: 1.4444 - accuracy: 0.
4681 - val_loss: 1.4600 - val_accuracy: 0.4775
Epoch 49/200
64/64 [==============================] - 13s 206ms/step - loss: 1.4691 - accuracy: 0.
4725 - val_loss: 1.4565 - val_accuracy: 0.4756
Epoch 50/200
64/64 [==============================] - 11s 173ms/step - loss: 1.4252 - accuracy: 0.
4839 - val_loss: 1.3794 - val_accuracy: 0.4919
Epoch 51/200
64/64 [==============================] - 11s 173ms/step - loss: 1.4422 - accuracy: 0.
4712 - val_loss: 1.4004 - val_accuracy: 0.4931
Epoch 52/200
64/64 [==============================] - 11s 169ms/step - loss: 1.4154 - accuracy: 0.
4802 - val_loss: 1.4474 - val_accuracy: 0.4675
Epoch 53/200
64/64 [==============================] - 11s 174ms/step - loss: 1.4332 - accuracy: 0.
4745 - val_loss: 1.4071 - val_accuracy: 0.4831
Epoch 54/200
64/64 [==============================] - 13s 200ms/step - loss: 1.4410 - accuracy: 0.
4714 - val_loss: 1.3953 - val_accuracy: 0.4863
Epoch 55/200
64/64 [==============================] - 11s 175ms/step - loss: 1.4612 - accuracy: 0.
4652 - val_loss: 1.4272 - val_accuracy: 0.4850
Epoch 56/200
64/64 [==============================] - 11s 174ms/step - loss: 1.4152 - accuracy: 0.
4841 - val_loss: 1.3921 - val_accuracy: 0.4863
Epoch 57/200
64/64 [==============================] - 12s 180ms/step - loss: 1.4215 - accuracy: 0.
4798 - val_loss: 1.4597 - val_accuracy: 0.4525
Epoch 58/200
64/64 [==============================] - 11s 177ms/step - loss: 1.4314 - accuracy: 0.
4816 - val_loss: 1.4329 - val_accuracy: 0.4819
Epoch 59/200
64/64 [==============================] - 12s 189ms/step - loss: 1.4141 - accuracy: 0.
4789 - val_loss: 1.3693 - val_accuracy: 0.4956
Epoch 60/200
64/64 [==============================] - 12s 181ms/step - loss: 1.4285 - accuracy: 0.
4764 - val_loss: 1.4468 - val_accuracy: 0.4569
```

```
Epoch 61/200
64/64 [==============================] - 11s 177ms/step - loss: 1.4545 - accuracy: 0.
4777 - val_loss: 1.4590 - val_accuracy: 0.4875
Epoch 62/200
64/64 [==============================] - 11s 174ms/step - loss: 1.4292 - accuracy: 0.
4755 - val_loss: 1.4582 - val_accuracy: 0.4769
Epoch 63/200
64/64 [==============================] - 11s 172ms/step - loss: 1.4327 - accuracy: 0.
4764 - val_loss: 1.4241 - val_accuracy: 0.4888
Epoch 64/200
64/64 [==============================] - 11s 173ms/step - loss: 1.4276 - accuracy: 0.
4764 - val_loss: 1.4093 - val_accuracy: 0.4975
Epoch 65/200
64/64 [==============================] - 11s 172ms/step - loss: 1.4179 - accuracy: 0.
4817 - val_loss: 1.3816 - val_accuracy: 0.4794
Epoch 66/200
64/64 [==============================] - 11s 173ms/step - loss: 1.4243 - accuracy: 0.
4828 - val_loss: 1.4437 - val_accuracy: 0.4756
Epoch 67/200
64/64 [==============================] - 11s 179ms/step - loss: 1.4115 - accuracy: 0.
4919 - val_loss: 1.4076 - val_accuracy: 0.4850
Epoch 68/200
64/64 [==============================] - 11s 169ms/step - loss: 1.4097 - accuracy: 0.
4855 - val_loss: 1.4242 - val_accuracy: 0.4850
Epoch 69/200
64/64 [==============================] - 11s 177ms/step - loss: 1.4277 - accuracy: 0.
4831 - val_loss: 1.3991 - val_accuracy: 0.4856
Epoch 70/200
64/64 [==============================] - 12s 186ms/step - loss: 1.4108 - accuracy: 0.
4855 - val_loss: 1.3844 - val_accuracy: 0.5063
Epoch 71/200
64/64 [==============================] - 11s 177ms/step - loss: 1.4139 - accuracy: 0.
4875 - val_loss: 1.4179 - val_accuracy: 0.4712
Epoch 72/200
64/64 [==============================] - 11s 170ms/step - loss: 1.4099 - accuracy: 0.
4812 - val_loss: 1.4309 - val_accuracy: 0.4769
Epoch 73/200
64/64 [==============================] - 11s 174ms/step - loss: 1.4154 - accuracy: 0.
4844 - val_loss: 1.4051 - val_accuracy: 0.5019
Epoch 74/200
64/64 [==============================] - 11s 171ms/step - loss: 1.4139 - accuracy: 0.
4897 - val_loss: 1.3899 - val_accuracy: 0.4925
Epoch 75/200
64/64 [==============================] - 12s 183ms/step - loss: 1.4106 - accuracy: 0.
4834 - val_loss: 1.4155 - val_accuracy: 0.4706
Epoch 76/200
64/64 [==============================] - 12s 184ms/step - loss: 1.4131 - accuracy: 0.
4897 - val_loss: 1.3527 - val_accuracy: 0.5150
Epoch 77/200
64/64 [==============================] - 11s 176ms/step - loss: 1.4026 - accuracy: 0.
4877 - val_loss: 1.4087 - val_accuracy: 0.5013
Epoch 78/200
64/64 [==============================] - 11s 178ms/step - loss: 1.3899 - accuracy: 0.
4892 - val_loss: 1.3688 - val_accuracy: 0.5000
Epoch 79/200
64/64 [==============================] - 11s 171ms/step - loss: 1.4158 - accuracy: 0.
4795 - val_loss: 1.3432 - val_accuracy: 0.5013
Epoch 80/200
64/64 [==============================] - 11s 168ms/step - loss: 1.3930 - accuracy: 0.
4963 - val_loss: 1.4180 - val_accuracy: 0.4881
```

```
Epoch 81/200
64/64 [==============================] - 11s 173ms/step - loss: 1.4040 - accuracy: 0.
4909 - val_loss: 1.3307 - val_accuracy: 0.5306
Epoch 82/200
64/64 [==============================] - 12s 183ms/step - loss: 1.3876 - accuracy: 0.
4889 - val_loss: 1.3649 - val_accuracy: 0.5225
Epoch 83/200
64/64 [==============================] - 12s 178ms/step - loss: 1.4190 - accuracy: 0.
4808 - val_loss: 1.3805 - val_accuracy: 0.5006
Epoch 84/200
64/64 [==============================] - 11s 174ms/step - loss: 1.3903 - accuracy: 0.
4970 - val_loss: 1.4115 - val_accuracy: 0.4900
Epoch 85/200
64/64 [==============================] - 11s 168ms/step - loss: 1.4000 - accuracy: 0.
4858 - val_loss: 1.3777 - val_accuracy: 0.5000
Epoch 86/200
64/64 [==============================] - 11s 171ms/step - loss: 1.4026 - accuracy: 0.
4916 - val_loss: 1.3747 - val_accuracy: 0.5013
Epoch 87/200
64/64 [==============================] - 11s 175ms/step - loss: 1.3745 - accuracy: 0.
5072 - val_loss: 1.4239 - val_accuracy: 0.4694
Epoch 88/200
64/64 [==============================] - 12s 188ms/step - loss: 1.3750 - accuracy: 0.
4942 - val_loss: 1.3580 - val_accuracy: 0.5006
Epoch 89/200
64/64 [==============================] - 11s 173ms/step - loss: 1.4022 - accuracy: 0.
5011 - val_loss: 1.4174 - val_accuracy: 0.4675
Epoch 90/200
64/64 [==============================] - 12s 190ms/step - loss: 1.3895 - accuracy: 0.
4948 - val_loss: 1.3850 - val_accuracy: 0.4906
Epoch 91/200
64/64 [==============================] - 12s 177ms/step - loss: 1.3908 - accuracy: 0.
4992 - val_loss: 1.3675 - val_accuracy: 0.5056
Epoch 92/200
64/64 [==============================] - 11s 173ms/step - loss: 1.3809 - accuracy: 0.
4922 - val_loss: 1.3842 - val_accuracy: 0.4925
Epoch 93/200
64/64 [==============================] - 11s 168ms/step - loss: 1.4025 - accuracy: 0.
4945 - val_loss: 1.3810 - val_accuracy: 0.5025
Epoch 94/200
64/64 [==============================] - 11s 175ms/step - loss: 1.3919 - accuracy: 0.
4892 - val_loss: 1.3911 - val_accuracy: 0.4988
Epoch 95/200
64/64 [==============================] - 11s 178ms/step - loss: 1.3860 - accuracy: 0.
4980 - val_loss: 1.3412 - val_accuracy: 0.5094
Epoch 96/200
64/64 [==============================] - 11s 175ms/step - loss: 1.3645 - accuracy: 0.
5011 - val_loss: 1.3267 - val_accuracy: 0.5063
Epoch 97/200
64/64 [==============================] - 11s 175ms/step - loss: 1.3879 - accuracy: 0.
4961 - val_loss: 1.3420 - val_accuracy: 0.5175
Epoch 98/200
64/64 [==============================] - 11s 174ms/step - loss: 1.3774 - accuracy: 0.
5019 - val_loss: 1.3973 - val_accuracy: 0.4762
Epoch 99/200
64/64 [==============================] - 11s 173ms/step - loss: 1.3937 - accuracy: 0.
4894 - val_loss: 1.3444 - val_accuracy: 0.5056
Epoch 100/200
64/64 [==============================] - 11s 176ms/step - loss: 1.3852 - accuracy: 0.
4988 - val_loss: 1.3331 - val_accuracy: 0.5088
```

```
Epoch 101/200
64/64 [==============================] - 11s 166ms/step - loss: 1.3751 - accuracy: 0.
5047 - val_loss: 1.3627 - val_accuracy: 0.5031
Epoch 102/200
64/64 [==============================] - 11s 177ms/step - loss: 1.3841 - accuracy: 0.
5005 - val_loss: 1.4000 - val_accuracy: 0.4913
Epoch 103/200
64/64 [==============================] - 11s 165ms/step - loss: 1.3782 - accuracy: 0.
4966 - val_loss: 1.3436 - val_accuracy: 0.5044
Epoch 104/200
64/64 [==============================] - 11s 170ms/step - loss: 1.3943 - accuracy: 0.
4947 - val_loss: 1.3968 - val_accuracy: 0.5063
Epoch 105/200
64/64 [==============================] - 11s 170ms/step - loss: 1.3726 - accuracy: 0.
5036 - val_loss: 1.3713 - val_accuracy: 0.4950
Epoch 106/200
64/64 [==============================] - 11s 171ms/step - loss: 1.3967 - accuracy: 0.
4967 - val_loss: 1.3898 - val_accuracy: 0.4850
Epoch 107/200
64/64 [==============================] - 11s 168ms/step - loss: 1.3835 - accuracy: 0.
4928 - val_loss: 1.3267 - val_accuracy: 0.5206
Epoch 108/200
64/64 [==============================] - 11s 170ms/step - loss: 1.3752 - accuracy: 0.
4955 - val_loss: 1.3824 - val_accuracy: 0.4787
Epoch 109/200
64/64 [==============================] - 11s 175ms/step - loss: 1.3725 - accuracy: 0.
4963 - val_loss: 1.3514 - val_accuracy: 0.5094
Epoch 110/200
64/64 [==============================] - 11s 172ms/step - loss: 1.3719 - accuracy: 0.
5025 - val_loss: 1.4152 - val_accuracy: 0.4837
Epoch 111/200
64/64 [==============================] - 11s 167ms/step - loss: 1.3755 - accuracy: 0.
4958 - val_loss: 1.3333 - val_accuracy: 0.5238
Epoch 112/200
64/64 [==============================] - 11s 168ms/step - loss: 1.3811 - accuracy: 0.
4970 - val_loss: 1.3361 - val_accuracy: 0.5219
Epoch 113/200
64/64 [==============================] - 11s 172ms/step - loss: 1.3943 - accuracy: 0.
4945 - val_loss: 1.3525 - val_accuracy: 0.5075
Epoch 114/200
64/64 [==============================] - 11s 166ms/step - loss: 1.3631 - accuracy: 0.
5113 - val_loss: 1.3994 - val_accuracy: 0.5188
Epoch 115/200
64/64 [==============================] - 11s 175ms/step - loss: 1.3739 - accuracy: 0.
5036 - val_loss: 1.3308 - val_accuracy: 0.5194
Epoch 116/200
64/64 [==============================] - 11s 172ms/step - loss: 1.3617 - accuracy: 0.
5070 - val_loss: 1.3475 - val_accuracy: 0.5100
Epoch 117/200
64/64 [==============================] - 11s 179ms/step - loss: 1.3765 - accuracy: 0.
4980 - val_loss: 1.3098 - val_accuracy: 0.5094
Epoch 118/200
64/64 [==============================] - 11s 169ms/step - loss: 1.3834 - accuracy: 0.
4942 - val_loss: 1.3508 - val_accuracy: 0.5169
Epoch 119/200
64/64 [==============================] - 11s 170ms/step - loss: 1.3745 - accuracy: 0.
4905 - val_loss: 1.3185 - val_accuracy: 0.5094
Epoch 120/200
64/64 [==============================] - 11s 168ms/step - loss: 1.3755 - accuracy: 0.
4975 - val_loss: 1.3689 - val_accuracy: 0.5044
```

```
Epoch 121/200
64/64 [==============================] - 12s 182ms/step - loss: 1.3773 - accuracy: 0.
4967 - val_loss: 1.3834 - val_accuracy: 0.4863
Epoch 122/200
64/64 [==============================] - 10s 163ms/step - loss: 1.3766 - accuracy: 0.
4967 - val_loss: 1.4306 - val_accuracy: 0.4750
Epoch 123/200
64/64 [==============================] - 12s 181ms/step - loss: 1.3955 - accuracy: 0.
4933 - val_loss: 1.3396 - val_accuracy: 0.5100
Epoch 124/200
64/64 [==============================] - 11s 172ms/step - loss: 1.3676 - accuracy: 0.
4952 - val_loss: 1.3085 - val_accuracy: 0.5206
Epoch 125/200
64/64 [==============================] - 11s 168ms/step - loss: 1.3672 - accuracy: 0.
5017 - val_loss: 1.3978 - val_accuracy: 0.4756
Epoch 126/200
64/64 [==============================] - 11s 168ms/step - loss: 1.3571 - accuracy: 0.
5045 - val_loss: 1.3608 - val_accuracy: 0.5025
Epoch 127/200
64/64 [==============================] - 11s 168ms/step - loss: 1.3898 - accuracy: 0.
4983 - val_loss: 1.3070 - val_accuracy: 0.5281
Epoch 128/200
64/64 [==============================] - 11s 170ms/step - loss: 1.3763 - accuracy: 0.
5034 - val_loss: 1.3675 - val_accuracy: 0.5094
Epoch 129/200
64/64 [==============================] - 11s 170ms/step - loss: 1.3682 - accuracy: 0.
5044 - val_loss: 1.3425 - val_accuracy: 0.5138
Epoch 130/200
64/64 [==============================] - 11s 164ms/step - loss: 1.3444 - accuracy: 0.
5123 - val_loss: 1.3400 - val_accuracy: 0.5038
Epoch 131/200
64/64 [==============================] - 10s 163ms/step - loss: 1.3641 - accuracy: 0.
5075 - val_loss: 1.2895 - val_accuracy: 0.5356
Epoch 132/200
64/64 [==============================] - 11s 165ms/step - loss: 1.3731 - accuracy: 0.
5053 - val_loss: 1.2982 - val_accuracy: 0.5150
Epoch 133/200
64/64 [==============================] - 11s 173ms/step - loss: 1.3528 - accuracy: 0.
5133 - val_loss: 1.3509 - val_accuracy: 0.4981
Epoch 134/200
64/64 [==============================] - 11s 172ms/step - loss: 1.3649 - accuracy: 0.
4988 - val_loss: 1.3484 - val_accuracy: 0.4944
Epoch 135/200
64/64 [==============================] - 11s 168ms/step - loss: 1.3706 - accuracy: 0.
4958 - val_loss: 1.3677 - val_accuracy: 0.5131
Epoch 136/200
64/64 [==============================] - 11s 167ms/step - loss: 1.3520 - accuracy: 0.
5097 - val_loss: 1.3097 - val_accuracy: 0.5419
Epoch 137/200
64/64 [==============================] - 11s 174ms/step - loss: 1.3720 - accuracy: 0.
5083 - val_loss: 1.3660 - val_accuracy: 0.5056
Epoch 138/200
64/64 [==============================] - 11s 168ms/step - loss: 1.3509 - accuracy: 0.
5086 - val_loss: 1.3408 - val_accuracy: 0.5144
Epoch 139/200
64/64 [==============================] - 11s 167ms/step - loss: 1.3530 - accuracy: 0.
5139 - val_loss: 1.4208 - val_accuracy: 0.4913
Epoch 140/200
64/64 [==============================] - 11s 169ms/step - loss: 1.3766 - accuracy: 0.
4975 - val_loss: 1.3557 - val_accuracy: 0.5113
```

```
Epoch 141/200
64/64 [==============================] - 11s 167ms/step - loss: 1.3744 - accuracy: 0.
4945 - val_loss: 1.4079 - val_accuracy: 0.4762
Epoch 142/200
64/64 [==============================] - 11s 170ms/step - loss: 1.3415 - accuracy: 0.
5100 - val_loss: 1.2817 - val_accuracy: 0.5288
Epoch 143/200
64/64 [==============================] - 11s 169ms/step - loss: 1.3670 - accuracy: 0.
4920 - val_loss: 1.3417 - val_accuracy: 0.5163
Epoch 144/200
64/64 [==============================] - 11s 172ms/step - loss: 1.3761 - accuracy: 0.
5013 - val_loss: 1.3516 - val_accuracy: 0.5069
Epoch 145/200
64/64 [==============================] - 11s 170ms/step - loss: 1.3570 - accuracy: 0.
5077 - val_loss: 1.3401 - val_accuracy: 0.5081
Epoch 146/200
64/64 [==============================] - 11s 166ms/step - loss: 1.3572 - accuracy: 0.
5059 - val_loss: 1.3245 - val_accuracy: 0.5144
Epoch 147/200
64/64 [==============================] - 11s 167ms/step - loss: 1.3767 - accuracy: 0.
4934 - val_loss: 1.3905 - val_accuracy: 0.4888
Epoch 148/200
64/64 [==============================] - 11s 165ms/step - loss: 1.3493 - accuracy: 0.
5094 - val_loss: 1.3228 - val_accuracy: 0.5263
Epoch 149/200
64/64 [==============================] - 10s 162ms/step - loss: 1.3565 - accuracy: 0.
5119 - val_loss: 1.3684 - val_accuracy: 0.5150
Epoch 150/200
64/64 [==============================] - 11s 172ms/step - loss: 1.3577 - accuracy: 0.
5097 - val_loss: 1.4068 - val_accuracy: 0.4913
Epoch 151/200
64/64 [==============================] - 11s 176ms/step - loss: 1.3461 - accuracy: 0.
5122 - val_loss: 1.3287 - val_accuracy: 0.5200
Epoch 152/200
64/64 [==============================] - 10s 161ms/step - loss: 1.3395 - accuracy: 0.
5125 - val_loss: 1.3023 - val_accuracy: 0.5244
Epoch 153/200
64/64 [==============================] - 10s 163ms/step - loss: 1.3459 - accuracy: 0.
5058 - val_loss: 1.3137 - val_accuracy: 0.5275
Epoch 154/200
64/64 [==============================] - 11s 172ms/step - loss: 1.3634 - accuracy: 0.
5039 - val_loss: 1.3859 - val_accuracy: 0.5063
Epoch 155/200
64/64 [==============================] - 11s 165ms/step - loss: 1.3619 - accuracy: 0.
5031 - val_loss: 1.4127 - val_accuracy: 0.4981
Epoch 156/200
64/64 [==============================] - 11s 168ms/step - loss: 1.3677 - accuracy: 0.
5041 - val_loss: 1.3248 - val_accuracy: 0.5213
Epoch 157/200
64/64 [==============================] - 11s 169ms/step - loss: 1.3184 - accuracy: 0.
5153 - val_loss: 1.3113 - val_accuracy: 0.5188
Epoch 158/200
64/64 [==============================] - 11s 175ms/step - loss: 1.3684 - accuracy: 0.
5063 - val_loss: 1.3399 - val_accuracy: 0.4994
Epoch 159/200
64/64 [==============================] - 11s 174ms/step - loss: 1.3470 - accuracy: 0.
5144 - val_loss: 1.3395 - val_accuracy: 0.5319
Epoch 160/200
64/64 [==============================] - 11s 176ms/step - loss: 1.3447 - accuracy: 0.
5191 - val_loss: 1.3478 - val_accuracy: 0.5081
```

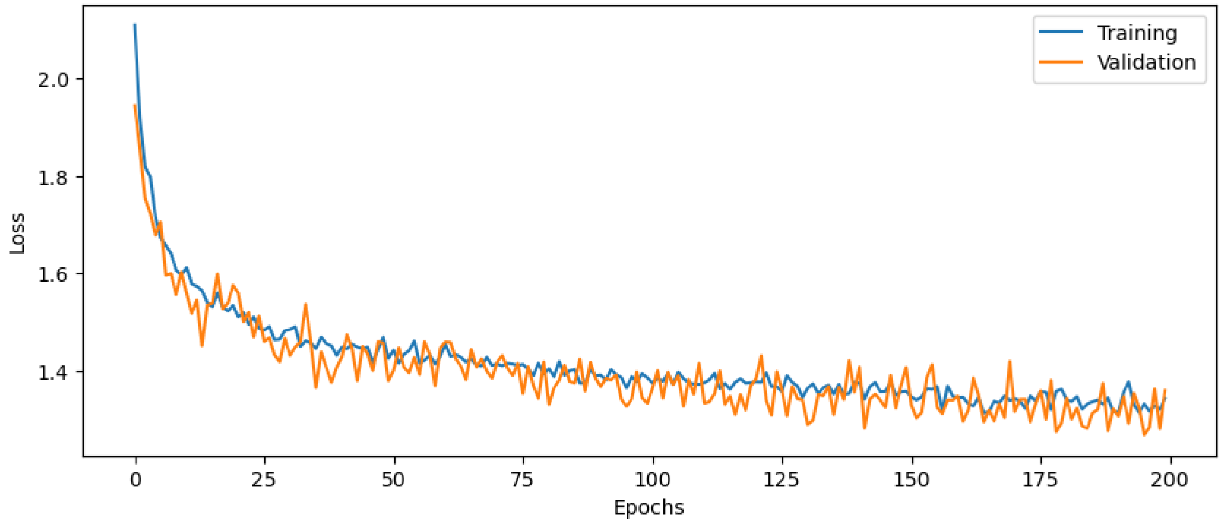
```
Epoch 161/200
64/64 [==============================] - 11s 175ms/step - loss: 1.3452 - accuracy: 0.
5138 - val_loss: 1.2962 - val_accuracy: 0.5450
Epoch 162/200
64/64 [==============================] - 10s 161ms/step - loss: 1.3322 - accuracy: 0.
5203 - val_loss: 1.3190 - val_accuracy: 0.5194
Epoch 163/200
64/64 [==============================] - 11s 171ms/step - loss: 1.3273 - accuracy: 0.
5191 - val_loss: 1.3854 - val_accuracy: 0.5094
Epoch 164/200
64/64 [==============================] - 11s 172ms/step - loss: 1.3450 - accuracy: 0.
5125 - val_loss: 1.3497 - val_accuracy: 0.5050
Epoch 165/200
64/64 [==============================] - 11s 174ms/step - loss: 1.3128 - accuracy: 0.
5231 - val_loss: 1.2948 - val_accuracy: 0.5337
Epoch 166/200
64/64 [==============================] - 12s 184ms/step - loss: 1.3153 - accuracy: 0.
5169 - val_loss: 1.3187 - val_accuracy: 0.5250
Epoch 167/200
64/64 [==============================] - 11s 176ms/step - loss: 1.3375 - accuracy: 0.
5150 - val_loss: 1.2967 - val_accuracy: 0.5325
Epoch 168/200
64/64 [==============================] - 11s 173ms/step - loss: 1.3341 - accuracy: 0.
5178 - val_loss: 1.3319 - val_accuracy: 0.5144
Epoch 169/200
64/64 [==============================] - 11s 169ms/step - loss: 1.3480 - accuracy: 0.
5202 - val_loss: 1.3036 - val_accuracy: 0.5188
Epoch 170/200
64/64 [==============================] - 11s 169ms/step - loss: 1.3388 - accuracy: 0.
5111 - val_loss: 1.4193 - val_accuracy: 0.4938
Epoch 171/200
64/64 [==============================] - 11s 164ms/step - loss: 1.3427 - accuracy: 0.
5202 - val_loss: 1.3158 - val_accuracy: 0.5088
Epoch 172/200
64/64 [==============================] - 11s 164ms/step - loss: 1.3379 - accuracy: 0.
5167 - val_loss: 1.3409 - val_accuracy: 0.5138
Epoch 173/200
64/64 [==============================] - 11s 179ms/step - loss: 1.3224 - accuracy: 0.
5177 - val_loss: 1.3411 - val_accuracy: 0.5075
Epoch 174/200
64/64 [==============================] - 11s 179ms/step - loss: 1.3483 - accuracy: 0.
5095 - val_loss: 1.2945 - val_accuracy: 0.5344
Epoch 175/200
64/64 [==============================] - 11s 165ms/step - loss: 1.3381 - accuracy: 0.
5172 - val_loss: 1.3309 - val_accuracy: 0.5200
Epoch 176/200
64/64 [==============================] - 11s 165ms/step - loss: 1.3569 - accuracy: 0.
5053 - val_loss: 1.3586 - val_accuracy: 0.5100
Epoch 177/200
64/64 [==============================] - 10s 164ms/step - loss: 1.3556 - accuracy: 0.
5048 - val_loss: 1.3002 - val_accuracy: 0.5256
Epoch 178/200
64/64 [==============================] - 11s 171ms/step - loss: 1.3197 - accuracy: 0.
5178 - val_loss: 1.3802 - val_accuracy: 0.5050
Epoch 179/200
64/64 [==============================] - 11s 172ms/step - loss: 1.3575 - accuracy: 0.
5089 - val_loss: 1.2743 - val_accuracy: 0.5400
Epoch 180/200
64/64 [==============================] - 11s 168ms/step - loss: 1.3621 - accuracy: 0.
5084 - val_loss: 1.2927 - val_accuracy: 0.5088
```

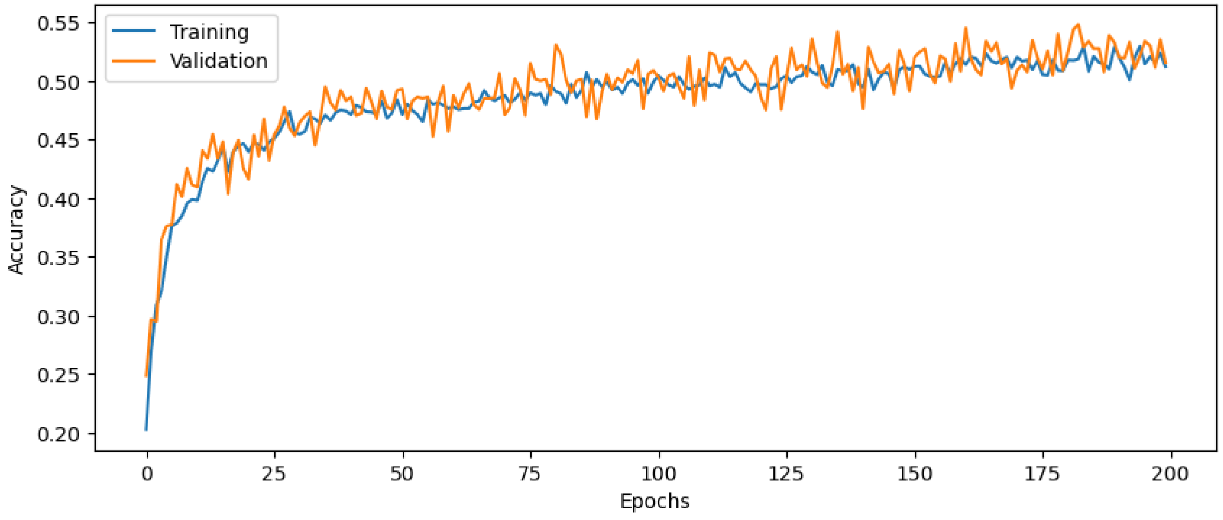
```
Epoch 181/200
64/64 [==============================] - 11s 168ms/step - loss: 1.3399 - accuracy: 0.
5178 - val_loss: 1.3434 - val_accuracy: 0.5250
Epoch 182/200
64/64 [==============================] - 11s 174ms/step - loss: 1.3372 - accuracy: 0.
5175 - val_loss: 1.3005 - val_accuracy: 0.5437
Epoch 183/200
64/64 [==============================] - 11s 168ms/step - loss: 1.3465 - accuracy: 0.
5186 - val_loss: 1.3229 - val_accuracy: 0.5481
Epoch 184/200
64/64 [==============================] - 11s 167ms/step - loss: 1.3207 - accuracy: 0.
5289 - val_loss: 1.2860 - val_accuracy: 0.5288
Epoch 185/200
64/64 [==============================] - 11s 174ms/step - loss: 1.3318 - accuracy: 0.
5080 - val_loss: 1.2812 - val_accuracy: 0.5337
Epoch 186/200
64/64 [==============================] - 11s 172ms/step - loss: 1.3363 - accuracy: 0.
5208 - val_loss: 1.3123 - val_accuracy: 0.5275
Epoch 187/200
64/64 [==============================] - 11s 170ms/step - loss: 1.3398 - accuracy: 0.
5158 - val_loss: 1.3200 - val_accuracy: 0.5275
Epoch 188/200
64/64 [==============================] - 11s 176ms/step - loss: 1.3313 - accuracy: 0.
5144 - val_loss: 1.3741 - val_accuracy: 0.5075
Epoch 189/200
64/64 [==============================] - 11s 173ms/step - loss: 1.3444 - accuracy: 0.
5100 - val_loss: 1.2764 - val_accuracy: 0.5387
Epoch 190/200
64/64 [==============================] - 11s 173ms/step - loss: 1.3159 - accuracy: 0.
5289 - val_loss: 1.3215 - val_accuracy: 0.5325
Epoch 191/200
64/64 [==============================] - 11s 168ms/step - loss: 1.3140 - accuracy: 0.
5198 - val_loss: 1.3066 - val_accuracy: 0.5200
Epoch 192/200
64/64 [==============================] - 11s 170ms/step - loss: 1.3555 - accuracy: 0.
5114 - val_loss: 1.3469 - val_accuracy: 0.5194
Epoch 193/200
64/64 [==============================] - 11s 170ms/step - loss: 1.3773 - accuracy: 0.
5005 - val_loss: 1.2919 - val_accuracy: 0.5331
Epoch 194/200
64/64 [==============================] - 11s 174ms/step - loss: 1.3306 - accuracy: 0.
5197 - val_loss: 1.3533 - val_accuracy: 0.5106
Epoch 195/200
64/64 [==============================] - 11s 177ms/step - loss: 1.3138 - accuracy: 0.
5295 - val_loss: 1.3236 - val_accuracy: 0.5213
Epoch 196/200
64/64 [==============================] - 11s 175ms/step - loss: 1.3323 - accuracy: 0.
5144 - val_loss: 1.2677 - val_accuracy: 0.5337
Epoch 197/200
64/64 [==============================] - 12s 184ms/step - loss: 1.3167 - accuracy: 0.
5206 - val_loss: 1.2829 - val_accuracy: 0.5300
Epoch 198/200
64/64 [==============================] - 12s 180ms/step - loss: 1.3280 - accuracy: 0.
5155 - val_loss: 1.3627 - val_accuracy: 0.5113
Epoch 199/200
64/64 [==============================] - 11s 171ms/step - loss: 1.3201 - accuracy: 0.
5236 - val_loss: 1.2807 - val_accuracy: 0.5350
Epoch 200/200
64/64 [==============================] - 11s 174ms/step - loss: 1.3433 - accuracy: 0.
5120 - val_loss: 1.3591 - val_accuracy: 0.5156
```

```
In [96]: # Check if there is still a big difference in accuracy for original and rotated test i

         # 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: 1.4695
Test accuracy: 0.4935
Test loss: 2.4050
Test accuracy: 0.2645
```

```
In [97]: # Plot the history from the training run
         plot_results(history6)
```





Part 20: Plot misclassified images

Lets plot some images where the CNN performed badly, these cells are already finished.

```
In [98]: # Find misclassified images
y_pred=model6.predict(Xtest)
y_pred=np.argmax(y_pred,axis=1)

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

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

```
63/63 [==============================] - 2s 21ms/step
```

```
In [99]: # 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_prec
plt.show()
```

Part 21: Testing on another size

Question 25: 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?

Answer:

No, we cannot apply this CNN to images of other size directly because the architecture of the CNN is designed to train the model with images of size 32 X 32. The size of the filters in a convolutional layer is choosen to match the size of input image. So changing input image size require change in the overall architecture of the CNN.

Question 26: 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?

Answer:

Yes, it is possible. Resize (compress or extend) an image to a fixed size before passing it through a pre-trained CNN.

Part 22: 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()`

Question 27: How many convolutional layers does ResNet50 have?

Answer:

ResNet50 has 50 convolutional layers.

Question 28: How many trainable parameters does the ResNet50 network have?

Answer:

It has 25,636,712 trainable parameters.

Question 29: What is the size of the images that ResNet50 expects as input?

Answer:

224 X 224

Question 30: Using the answer to question 28, explain why the second derivative is seldom used when training deep networks.

Answer:

The number of trainable parameters in deep neural networks can be very large (millions) number of parameters, depending on the size and complexity of the network. Calculating the second derivative of the loss function with respect to all of these parameters would need a huge amount of memory and computational resources.

Apply the pre-trained CNN 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 laboration.

See https://keras.io/api/applications/ and https://keras.io/api/applications/resnet/#resnet50-function

Useful functions

`image.load_img` in tensorflow.keras.preprocessing

`image.img_to_array` in tensorflow.keras.preprocessing

`ResNet50` in tensorflow.keras.applications.resnet50 `preprocess_input` in tensorflow.keras.applications.resnet50

`decode_predictions` in tensorflow.keras.applications.resnet50

`expand_dims` in numpy

In [100…

```
from keras.applications import ResNet50
model = ResNet50(weights='imagenet')
model.summary()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/re
snet/resnet50_weights_tf_dim_ordering_tf_kernels.h5
102967424/102967424 [==============================] - 12s 0us/step
Model: "resnet50"
_____________________________________________________________________________________
_____________
 Layer (type)                   Output Shape         Param #     Connected to
=====================================================================================
=============
 input_1 (InputLayer)           [(None, 224, 224, 3  0           []
                                )]

 conv1_pad (ZeroPadding2D)      (None, 230, 230, 3)  0           ['input_1[0][0]']

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

 conv1_bn (BatchNormalization)  (None, 112, 112, 64  256         ['conv1_conv[0][0]']
                                )

 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 (Conv2D)   (None, 56, 56, 64)   4160        ['pool1_pool[0][0]']

 conv2_block1_1_bn (BatchNormal  (None, 56, 56, 64)  256         ['conv2_block1_1_con
v[0][0]']
 ization)

 conv2_block1_1_relu (Activatio  (None, 56, 56, 64)  0           ['conv2_block1_1_bn
[0][0]']
 n)

 conv2_block1_2_conv (Conv2D)   (None, 56, 56, 64)   36928       ['conv2_block1_1_rel
u[0][0]']

 conv2_block1_2_bn (BatchNormal  (None, 56, 56, 64)  256         ['conv2_block1_2_con
v[0][0]']
 ization)

 conv2_block1_2_relu (Activatio  (None, 56, 56, 64)  0           ['conv2_block1_2_bn
[0][0]']
 n)

 conv2_block1_0_conv (Conv2D)   (None, 56, 56, 256)  16640       ['pool1_pool[0][0]']

 conv2_block1_3_conv (Conv2D)   (None, 56, 56, 256)  16640       ['conv2_block1_2_rel
u[0][0]']

 conv2_block1_0_bn (BatchNormal  (None, 56, 56, 256)  1024       ['conv2_block1_0_con
v[0][0]']
 ization)

 conv2_block1_3_bn (BatchNormal  (None, 56, 56, 256)  1024       ['conv2_block1_3_con
v[0][0]']
```

```
 ization)

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

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

 conv2_block2_1_conv (Conv2D)   (None, 56, 56, 64)   16448       ['conv2_block1_out
[0][0]']

 conv2_block2_1_bn (BatchNormal  (None, 56, 56, 64)  256         ['conv2_block2_1_con
v[0][0]']
 ization)

 conv2_block2_1_relu (Activatio  (None, 56, 56, 64)  0           ['conv2_block2_1_bn
[0][0]']
 n)

 conv2_block2_2_conv (Conv2D)   (None, 56, 56, 64)   36928       ['conv2_block2_1_rel
u[0][0]']

 conv2_block2_2_bn (BatchNormal  (None, 56, 56, 64)  256         ['conv2_block2_2_con
v[0][0]']
 ization)

 conv2_block2_2_relu (Activatio  (None, 56, 56, 64)  0           ['conv2_block2_2_bn
[0][0]']
 n)

 conv2_block2_3_conv (Conv2D)   (None, 56, 56, 256)  16640       ['conv2_block2_2_rel
u[0][0]']

 conv2_block2_3_bn (BatchNormal  (None, 56, 56, 256)  1024       ['conv2_block2_3_con
v[0][0]']
 ization)

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

 conv2_block2_out (Activation)  (None, 56, 56, 256)  0           ['conv2_block2_add
[0][0]']

 conv2_block3_1_conv (Conv2D)   (None, 56, 56, 64)   16448       ['conv2_block2_out
[0][0]']

 conv2_block3_1_bn (BatchNormal  (None, 56, 56, 64)  256         ['conv2_block3_1_con
v[0][0]']
 ization)

 conv2_block3_1_relu (Activatio  (None, 56, 56, 64)  0           ['conv2_block3_1_bn
[0][0]']
 n)

 conv2_block3_2_conv (Conv2D)   (None, 56, 56, 64)   36928       ['conv2_block3_1_rel
u[0][0]']
```

```
 conv2_block3_2_bn (BatchNormal  (None, 56, 56, 64)  256         ['conv2_block3_2_con
v[0][0]']
 ization)

 conv2_block3_2_relu (Activatio  (None, 56, 56, 64)  0           ['conv2_block3_2_bn
[0][0]']
 n)

 conv2_block3_3_conv (Conv2D)   (None, 56, 56, 256)  16640       ['conv2_block3_2_rel
u[0][0]']

 conv2_block3_3_bn (BatchNormal  (None, 56, 56, 256)  1024       ['conv2_block3_3_con
v[0][0]']
 ization)

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

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

 conv3_block1_1_conv (Conv2D)   (None, 28, 28, 128)  32896       ['conv2_block3_out
[0][0]']

 conv3_block1_1_bn (BatchNormal  (None, 28, 28, 128)  512        ['conv3_block1_1_con
v[0][0]']
 ization)

 conv3_block1_1_relu (Activatio  (None, 28, 28, 128)  0          ['conv3_block1_1_bn
[0][0]']
 n)

 conv3_block1_2_conv (Conv2D)   (None, 28, 28, 128)  147584      ['conv3_block1_1_rel
u[0][0]']

 conv3_block1_2_bn (BatchNormal  (None, 28, 28, 128)  512        ['conv3_block1_2_con
v[0][0]']
 ization)

 conv3_block1_2_relu (Activatio  (None, 28, 28, 128)  0          ['conv3_block1_2_bn
[0][0]']
 n)

 conv3_block1_0_conv (Conv2D)   (None, 28, 28, 512)  131584      ['conv2_block3_out
[0][0]']

 conv3_block1_3_conv (Conv2D)   (None, 28, 28, 512)  66048       ['conv3_block1_2_rel
u[0][0]']

 conv3_block1_0_bn (BatchNormal  (None, 28, 28, 512)  2048       ['conv3_block1_0_con
v[0][0]']
 ization)

 conv3_block1_3_bn (BatchNormal  (None, 28, 28, 512)  2048       ['conv3_block1_3_con
v[0][0]']
 ization)
 conv3_block1_add (Add)         (None, 28, 28, 512)  0           ['conv3_block1_0_bn
[0][0]',
                                                                  'conv3_block1_3_bn
[0][0]']

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

 conv3_block2_1_conv (Conv2D)   (None, 28, 28, 128)  65664       ['conv3_block1_out
[0][0]']

 conv3_block2_1_bn (BatchNormal  (None, 28, 28, 128)  512        ['conv3_block2_1_con
v[0][0]']
 ization)

 conv3_block2_1_relu (Activatio  (None, 28, 28, 128)  0          ['conv3_block2_1_bn
[0][0]']
 n)

 conv3_block2_2_conv (Conv2D)   (None, 28, 28, 128)  147584      ['conv3_block2_1_rel
u[0][0]']

 conv3_block2_2_bn (BatchNormal  (None, 28, 28, 128)  512        ['conv3_block2_2_con
v[0][0]']
 ization)

 conv3_block2_2_relu (Activatio  (None, 28, 28, 128)  0          ['conv3_block2_2_bn
[0][0]']
 n)

 conv3_block2_3_conv (Conv2D)   (None, 28, 28, 512)  66048       ['conv3_block2_2_rel
u[0][0]']

 conv3_block2_3_bn (BatchNormal  (None, 28, 28, 512)  2048       ['conv3_block2_3_con
v[0][0]']
 ization)

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

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

 conv3_block3_1_conv (Conv2D)   (None, 28, 28, 128)  65664       ['conv3_block2_out
[0][0]']

 conv3_block3_1_bn (BatchNormal  (None, 28, 28, 128)  512        ['conv3_block3_1_con
v[0][0]']
 ization)

 conv3_block3_1_relu (Activatio  (None, 28, 28, 128)  0          ['conv3_block3_1_bn
[0][0]']
 n)

 conv3_block3_2_conv (Conv2D)   (None, 28, 28, 128)  147584      ['conv3_block3_1_rel
u[0][0]']

 conv3_block3_2_bn (BatchNormal  (None, 28, 28, 128)  512        ['conv3_block3_2_con
v[0][0]']
 ization)

 conv3_block3_2_relu (Activatio  (None, 28, 28, 128)  0           ['conv3_block3_2_bn
[0][0]']
 n)

 conv3_block3_3_conv (Conv2D)   (None, 28, 28, 512)  66048       ['conv3_block3_2_rel
u[0][0]']

 conv3_block3_3_bn (BatchNormal  (None, 28, 28, 512)  2048       ['conv3_block3_3_con
v[0][0]']
 ization)

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

[0][0]']

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

 conv3_block4_1_conv (Conv2D)   (None, 28, 28, 128)  65664       ['conv3_block3_out
[0][0]']

 conv3_block4_1_bn (BatchNormal  (None, 28, 28, 128)  512        ['conv3_block4_1_con
v[0][0]']
 ization)

 conv3_block4_1_relu (Activatio  (None, 28, 28, 128)  0           ['conv3_block4_1_bn
[0][0]']
 n)

 conv3_block4_2_conv (Conv2D)   (None, 28, 28, 128)  147584      ['conv3_block4_1_rel
u[0][0]']

 conv3_block4_2_bn (BatchNormal  (None, 28, 28, 128)  512        ['conv3_block4_2_con
v[0][0]']
 ization)

 conv3_block4_2_relu (Activatio  (None, 28, 28, 128)  0           ['conv3_block4_2_bn
[0][0]']
 n)

 conv3_block4_3_conv (Conv2D)   (None, 28, 28, 512)  66048       ['conv3_block4_2_rel
u[0][0]']

 conv3_block4_3_bn (BatchNormal  (None, 28, 28, 512)  2048       ['conv3_block4_3_con
v[0][0]']
 ization)

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

[0][0]']

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

 conv4_block1_1_conv (Conv2D)   (None, 14, 14, 256)  131328      ['conv3_block4_out
```

```
[0][0]']

 conv4_block1_1_bn (BatchNormal  (None, 14, 14, 256)  1024        ['conv4_block1_1_con
v[0][0]']
 ization)

 conv4_block1_1_relu (Activatio  (None, 14, 14, 256)  0           ['conv4_block1_1_bn
[0][0]']
 n)

 conv4_block1_2_conv (Conv2D)   (None, 14, 14, 256)  590080      ['conv4_block1_1_rel
u[0][0]']

 conv4_block1_2_bn (BatchNormal  (None, 14, 14, 256)  1024        ['conv4_block1_2_con
v[0][0]']
 ization)

 conv4_block1_2_relu (Activatio  (None, 14, 14, 256)  0           ['conv4_block1_2_bn
[0][0]']
 n)

 conv4_block1_0_conv (Conv2D)   (None, 14, 14, 1024  525312      ['conv3_block4_out
[0][0]']
                                )

 conv4_block1_3_conv (Conv2D)   (None, 14, 14, 1024  263168      ['conv4_block1_2_rel
u[0][0]']
                                )

 conv4_block1_0_bn (BatchNormal  (None, 14, 14, 1024  4096        ['conv4_block1_0_con
v[0][0]']
 ization)                       )

 conv4_block1_3_bn (BatchNormal  (None, 14, 14, 1024  4096        ['conv4_block1_3_con
v[0][0]']
 ization)                       )

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

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

 conv4_block2_1_conv (Conv2D)   (None, 14, 14, 256)  262400      ['conv4_block1_out
[0][0]']

 conv4_block2_1_bn (BatchNormal  (None, 14, 14, 256)  1024        ['conv4_block2_1_con
v[0][0]']
 ization)

 conv4_block2_1_relu (Activatio  (None, 14, 14, 256)  0           ['conv4_block2_1_bn
[0][0]']
 n)

 conv4_block2_2_conv (Conv2D)   (None, 14, 14, 256)  590080      ['conv4_block2_1_rel
u[0][0]']
```

```
 conv4_block2_2_bn (BatchNormal  (None, 14, 14, 256)  1024        ['conv4_block2_2_con
v[0][0]']
 ization)

 conv4_block2_2_relu (Activatio  (None, 14, 14, 256)  0           ['conv4_block2_2_bn
[0][0]']
 n)

 conv4_block2_3_conv (Conv2D)   (None, 14, 14, 1024  263168      ['conv4_block2_2_rel
u[0][0]']
                                )

 conv4_block2_3_bn (BatchNormal  (None, 14, 14, 1024  4096        ['conv4_block2_3_con
v[0][0]']
 ization)                       )

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

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

 conv4_block3_1_conv (Conv2D)   (None, 14, 14, 256)  262400      ['conv4_block2_out
[0][0]']

 conv4_block3_1_bn (BatchNormal  (None, 14, 14, 256)  1024        ['conv4_block3_1_con
v[0][0]']
 ization)

 conv4_block3_1_relu (Activatio  (None, 14, 14, 256)  0           ['conv4_block3_1_bn
[0][0]']
 n)

 conv4_block3_2_conv (Conv2D)   (None, 14, 14, 256)  590080      ['conv4_block3_1_rel
u[0][0]']

 conv4_block3_2_bn (BatchNormal  (None, 14, 14, 256)  1024        ['conv4_block3_2_con
v[0][0]']
 ization)

 conv4_block3_2_relu (Activatio  (None, 14, 14, 256)  0           ['conv4_block3_2_bn
[0][0]']
 n)

 conv4_block3_3_conv (Conv2D)   (None, 14, 14, 1024  263168      ['conv4_block3_2_rel
u[0][0]']
                                )

 conv4_block3_3_bn (BatchNormal  (None, 14, 14, 1024  4096        ['conv4_block3_3_con
v[0][0]']
 ization)                       )

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

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

 conv4_block4_1_conv (Conv2D)   (None, 14, 14, 256)  262400      ['conv4_block3_out
[0][0]']

 conv4_block4_1_bn (BatchNormal  (None, 14, 14, 256)  1024       ['conv4_block4_1_con
v[0][0]']
 ization)

 conv4_block4_1_relu (Activatio  (None, 14, 14, 256)  0          ['conv4_block4_1_bn
[0][0]']
 n)

 conv4_block4_2_conv (Conv2D)   (None, 14, 14, 256)  590080      ['conv4_block4_1_rel
u[0][0]']

 conv4_block4_2_bn (BatchNormal  (None, 14, 14, 256)  1024       ['conv4_block4_2_con
v[0][0]']
 ization)

 conv4_block4_2_relu (Activatio  (None, 14, 14, 256)  0          ['conv4_block4_2_bn
[0][0]']
 n)

 conv4_block4_3_conv (Conv2D)   (None, 14, 14, 1024  263168      ['conv4_block4_2_rel
u[0][0]']
                                )

 conv4_block4_3_bn (BatchNormal  (None, 14, 14, 1024  4096       ['conv4_block4_3_con
v[0][0]']
 ization)                       )

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

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

 conv4_block5_1_conv (Conv2D)   (None, 14, 14, 256)  262400      ['conv4_block4_out
[0][0]']

 conv4_block5_1_bn (BatchNormal  (None, 14, 14, 256)  1024       ['conv4_block5_1_con
v[0][0]']
 ization)

 conv4_block5_1_relu (Activatio  (None, 14, 14, 256)  0          ['conv4_block5_1_bn
[0][0]']
 n)

 conv4_block5_2_conv (Conv2D)   (None, 14, 14, 256)  590080      ['conv4_block5_1_rel
u[0][0]']

 conv4_block5_2_bn (BatchNormal  (None, 14, 14, 256)  1024       ['conv4_block5_2_con
v[0][0]']
 ization)
 conv4_block5_2_relu (Activatio  (None, 14, 14, 256)  0           ['conv4_block5_2_bn
[0][0]']
 n)

 conv4_block5_3_conv (Conv2D)   (None, 14, 14, 1024  263168      ['conv4_block5_2_rel
u[0][0]']
                                )

 conv4_block5_3_bn (BatchNormal  (None, 14, 14, 1024  4096        ['conv4_block5_3_con
v[0][0]']
 ization)                       )

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

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

 conv4_block6_1_conv (Conv2D)   (None, 14, 14, 256)  262400      ['conv4_block5_out
[0][0]']

 conv4_block6_1_bn (BatchNormal  (None, 14, 14, 256)  1024        ['conv4_block6_1_con
v[0][0]']
 ization)

 conv4_block6_1_relu (Activatio  (None, 14, 14, 256)  0           ['conv4_block6_1_bn
[0][0]']
 n)

 conv4_block6_2_conv (Conv2D)   (None, 14, 14, 256)  590080      ['conv4_block6_1_rel
u[0][0]']

 conv4_block6_2_bn (BatchNormal  (None, 14, 14, 256)  1024        ['conv4_block6_2_con
v[0][0]']
 ization)

 conv4_block6_2_relu (Activatio  (None, 14, 14, 256)  0           ['conv4_block6_2_bn
[0][0]']
 n)

 conv4_block6_3_conv (Conv2D)   (None, 14, 14, 1024  263168      ['conv4_block6_2_rel
u[0][0]']
                                )

 conv4_block6_3_bn (BatchNormal  (None, 14, 14, 1024  4096        ['conv4_block6_3_con
v[0][0]']
 ization)                       )

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

 conv4_block6_out (Activation)  (None, 14, 14, 1024  0           ['conv4_block6_add
[0][0]']
                                )
 conv5_block1_1_conv (Conv2D)   (None, 7, 7, 512)    524800      ['conv4_block6_out
[0][0]']

 conv5_block1_1_bn (BatchNormal  (None, 7, 7, 512)   2048        ['conv5_block1_1_con
v[0][0]']
 ization)

 conv5_block1_1_relu (Activatio  (None, 7, 7, 512)   0           ['conv5_block1_1_bn
[0][0]']
 n)

 conv5_block1_2_conv (Conv2D)   (None, 7, 7, 512)    2359808     ['conv5_block1_1_rel
u[0][0]']

 conv5_block1_2_bn (BatchNormal  (None, 7, 7, 512)   2048        ['conv5_block1_2_con
v[0][0]']
 ization)

 conv5_block1_2_relu (Activatio  (None, 7, 7, 512)   0           ['conv5_block1_2_bn
[0][0]']
 n)

 conv5_block1_0_conv (Conv2D)   (None, 7, 7, 2048)   2099200     ['conv4_block6_out
[0][0]']

 conv5_block1_3_conv (Conv2D)   (None, 7, 7, 2048)   1050624     ['conv5_block1_2_rel
u[0][0]']

 conv5_block1_0_bn (BatchNormal  (None, 7, 7, 2048)  8192        ['conv5_block1_0_con
v[0][0]']
 ization)

 conv5_block1_3_bn (BatchNormal  (None, 7, 7, 2048)  8192        ['conv5_block1_3_con
v[0][0]']
 ization)

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

                                                                  'conv5_block1_3_bn
[0][0]']

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

 conv5_block2_1_conv (Conv2D)   (None, 7, 7, 512)    1049088     ['conv5_block1_out
[0][0]']

 conv5_block2_1_bn (BatchNormal  (None, 7, 7, 512)   2048        ['conv5_block2_1_con
v[0][0]']
 ization)

 conv5_block2_1_relu (Activatio  (None, 7, 7, 512)   0           ['conv5_block2_1_bn
[0][0]']
 n)

 conv5_block2_2_conv (Conv2D)   (None, 7, 7, 512)    2359808     ['conv5_block2_1_rel
u[0][0]']

 conv5_block2_2_bn (BatchNormal  (None, 7, 7, 512)   2048        ['conv5_block2_2_con
v[0][0]']
 ization)

 conv5_block2_2_relu (Activatio  (None, 7, 7, 512)   0           ['conv5_block2_2_bn
[0][0]']
 n)

 conv5_block2_3_conv (Conv2D)   (None, 7, 7, 2048)   1050624     ['conv5_block2_2_rel
u[0][0]']

 conv5_block2_3_bn (BatchNormal  (None, 7, 7, 2048)  8192        ['conv5_block2_3_con
v[0][0]']
 ization)

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

[0][0]']

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

 conv5_block3_1_conv (Conv2D)   (None, 7, 7, 512)    1049088     ['conv5_block2_out
[0][0]']

 conv5_block3_1_bn (BatchNormal  (None, 7, 7, 512)   2048        ['conv5_block3_1_con
v[0][0]']
 ization)

 conv5_block3_1_relu (Activatio  (None, 7, 7, 512)   0           ['conv5_block3_1_bn
[0][0]']
 n)

 conv5_block3_2_conv (Conv2D)   (None, 7, 7, 512)    2359808     ['conv5_block3_1_rel
u[0][0]']

 conv5_block3_2_bn (BatchNormal  (None, 7, 7, 512)   2048        ['conv5_block3_2_con
v[0][0]']
 ization)

 conv5_block3_2_relu (Activatio  (None, 7, 7, 512)   0           ['conv5_block3_2_bn
[0][0]']
 n)

 conv5_block3_3_conv (Conv2D)   (None, 7, 7, 2048)   1050624     ['conv5_block3_2_rel
u[0][0]']

 conv5_block3_3_bn (BatchNormal  (None, 7, 7, 2048)  8192        ['conv5_block3_3_con
v[0][0]']
 ization)

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

[0][0]']

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

 avg_pool (GlobalAveragePooling  (None, 2048)        0           ['conv5_block3_out
[0][0]']
 2D)

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

==================================================================================================
Total params: 25,636,712
Trainable params: 25,583,592
Non-trainable params: 53,120
__________________________________________________________________________________________________
```

In [1]:

```
# Your code for using pre-trained ResNet 50 on 5 color images of your choice.
# The preprocessing should transform the image to a size that is expected by the CNN.

from tensorflow.keras.applications.resnet50 import ResNet50
from tensorflow.keras.applications.resnet50 import decode_predictions
from tensorflow.keras.applications.resnet50 import preprocess_input
from tensorflow.keras.preprocessing import image
import numpy as np

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

# Define the target size for the ResNet50 model
target_size = (224, 224)

# Load and preprocess the image
images = ['cat.jpg','car.jpg','elephant.jpg','horse.jpg','aeroplane.jpg']

for img_name in images:
    img_path = img_name
    img = image.load_img(img_path, target_size=target_size)
    img = image.img_to_array(img)
    img = np.expand_dims(img, axis=0)
    img = preprocess_input(img)
    preds = model.predict(img)
    decoded_preds = decode_predictions(preds, top=1)[0]
    label = decoded_preds[0][1]
    print('Predicted label: {}'.format(label))
```

```
1/1 [==============================] - 6s 6s/step
Predicted label: Egyptian_cat
1/1 [==============================] - 0s 445ms/step
Predicted label: minivan
1/1 [==============================] - 0s 366ms/step
Predicted label: African_elephant
1/1 [==============================] - 0s 337ms/step
Predicted label: horse_cart
1/1 [==============================] - 0s 331ms/step
Predicted label: airliner
```