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 [2]:

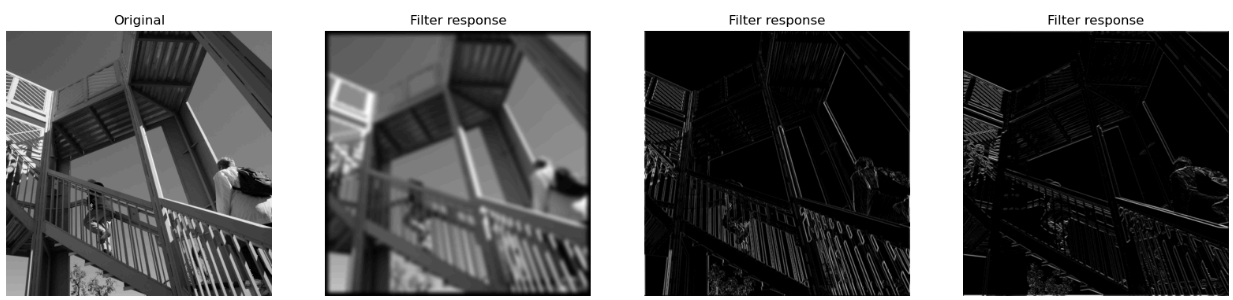
```
# This cell is finished

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\PC\AppData\Local\Temp\ipykernel_46420\2994295117.py:8: DeprecationWarning: s
cipy.misc.ascent has been deprecated in SciPy v1.10.0; and will be completely removed
in SciPy v1.12.0. Dataset methods have moved into the scipy.datasets module. Use scip
y.datasets.ascent instead.
  image = misc.ascent()
```

In [3]:
```
# Perform convolution using the function 'convolve2d' for the different filters
filterResponseGauss = signal.convolve2d(image,gaussFilter)
filterResponseSobelX = signal.convolve2d(image,sobelX)
filterResponseSobelY = signal.convolve2d(image,sobelY)
```

In [4]:
```
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?

- Gaussian filter blurs the image, SobelX enhances the vertical patterns, and SobelY enhances the horizontal patterns in the image.

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?

- 512x512
- There is a only one channel in original photo because of it is a greyscale image. And, colour images have 3 channels.

Question 3: What is the size of the different filters?

- Gaussian: 15x15
- SobelX : 3x3
- SobelY : 3x3

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

- If 'same' mode is used, filter response has the same size as the 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?

- mode = 'valid':The output consists only of those elements that do not rely on the zero-padding. In ‘valid’ mode, either in 1 or in2 must be at least as large as the other in every dimension.

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

- In CNNs with many layers, if each layer uses 'valid' filters, then will reduce the dimensions of input when we have many layers

In [5]:

```
# Your code for checking sizes of image and filter responses

print(f"size of original image: {image.shape}")

# Size of different filters
print(f"size of gaussian filter: {gaussFilter.shape}\nsize of SobelX filter: {sobelX.s

#when we change the mode

print(signal.convolve2d(image,gaussFilter,mode='same').shape)
print(signal.convolve2d(image,gaussFilter,mode='valid').shape)
```

```
size of original image: (512, 512)
size of gaussian filter: (15, 15)
size of SobelX filter: (3, 3)
size of SobelY filter: (3, 3)
(512, 512)
(498, 498)
```

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 [6]:
```
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)
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
Cell In[6], line 17
     15 # Allow growth of GPU memory, otherwise it will always look like all the memo
ry is being used
     16 physical_devices = tf.config.experimental.list_physical_devices('GPU')
---> 17 tf.config.experimental.set_memory_growth(physical_devices[0], True)

IndexError: list index out of range
```

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 ?

- Filters used for color images have an extra dimension for the different channels of the image. (RGB channels)

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?

- Yes, the Conv2D layer in convolutional neural networks (CNNs) performs a standard 2D convolution operation, similar to the convolution operation performed by the signal.convolve2d function. in 'Conv2D', it's using crosscorrelation, instead of standard 2D convolution

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.

- yes, graphics card is good at compute massive simples tasks parallelly, compared to the CPU, which has more powerful but much less cores than GPU

Part 5: Load data

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

In [7]:

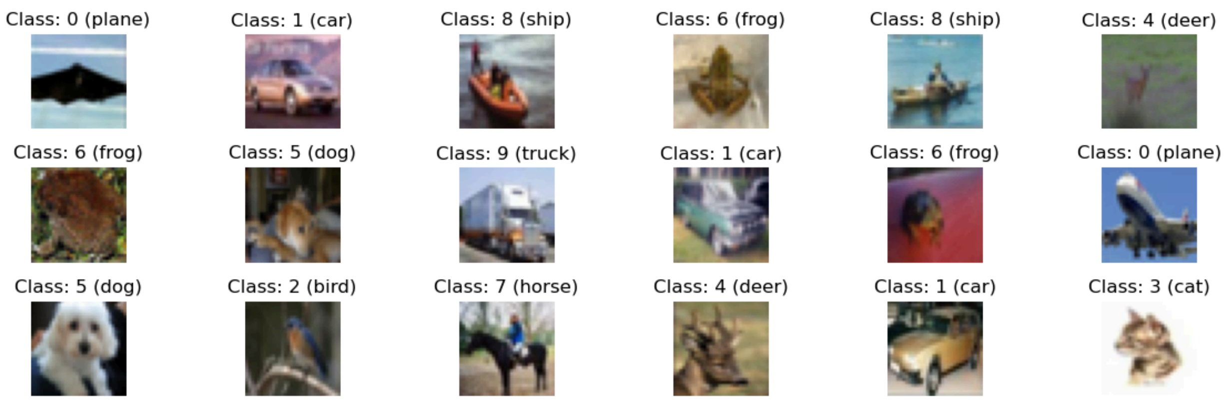
```
from keras.datasets import cifar10
import numpy as np

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 [8]:
```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 [9]:
```python
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=0.3, random_st
# Print the size of training data, validation data and test data
for set in (Xtrain, Xval, Ytrain, Yval):
    print(f'{set.shape}')
```

```
(7000, 32, 32, 3)
(3000, 32, 32, 3)
(7000, 1)
(3000, 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 [10]:
```python
# Convert datatype for Xtrain, Xval, Xtest, to float32
Xtrain = Xtrain.astype('float32')
Xval = Xval.astype('float32')
Xtest = Xtest.astype('float32')

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

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 [32]:
```python
from tensorflow.keras.utils import to_categorical

# Print shapes before converting the labels
print(f"size of labels before converting:\nYtrain: {Ytrain.shape}\nYval: {Yval.shape}\

# Your code for converting Ytrain, Yval, Ytest to categorical
Ytrain = to_categorical(Ytrain, num_classes=len(classes))
Yval = to_categorical(Yval, num_classes=len(classes))
Ytest = to_categorical(Ytest, num_classes=len(classes))

# Print shapes after converting the labels
print('after converting:')

print(f'Ytrain.shape = {Ytrain.shape}')
print(f'Yval.shape = {Yval.shape}')
print(f'Ytest.shape = {Ytest.shape}')
```

```
size of labels before converting:
Ytrain: (10000, 10)
Yval: (3000, 10)
Ytest: (2000, 10)
after converting:
Ytrain.shape = (10000, 10, 10)
Yval.shape = (3000, 10, 10)
Ytest.shape = (2000, 10, 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 [12]:

```
from keras.models import Sequential, Model
from keras.layers import Input, Conv2D, BatchNormalization, MaxPooling2D, Flatten, Der
from tensorflow.keras.optimizers import Adam
from keras.losses import CategoricalCrossentropy

# 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()
```

```python
    # Add first convolutional layer to the model, requires input shape
    model.add(Conv2D(filters = n_filters, kernel_size = 3, padding = "same", activatic
    model.add(BatchNormalization())
    model.add(MaxPooling2D(pool_size=(2,2)))


    # Add remaining convolutional layers to the model, the number of filters should ir
    for i in range(n_conv_layers-1):
        n_filters *= 2
        model.add(Conv2D(filters = n_filters, kernel_size = 3, padding = "same", activ
        model.add(BatchNormalization())
        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'))
        model.add(BatchNormalization())

        if use_dropout:
            if use_dropout == True:
                use_dropout = 0.5
            model.add(Dropout(rate = use_dropout))

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

    # Compile model
    model.compile(optimizer = Adam(learning_rate = learning_rate), loss = CategoricalC

    return model
```

In [13]:

```python
# 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

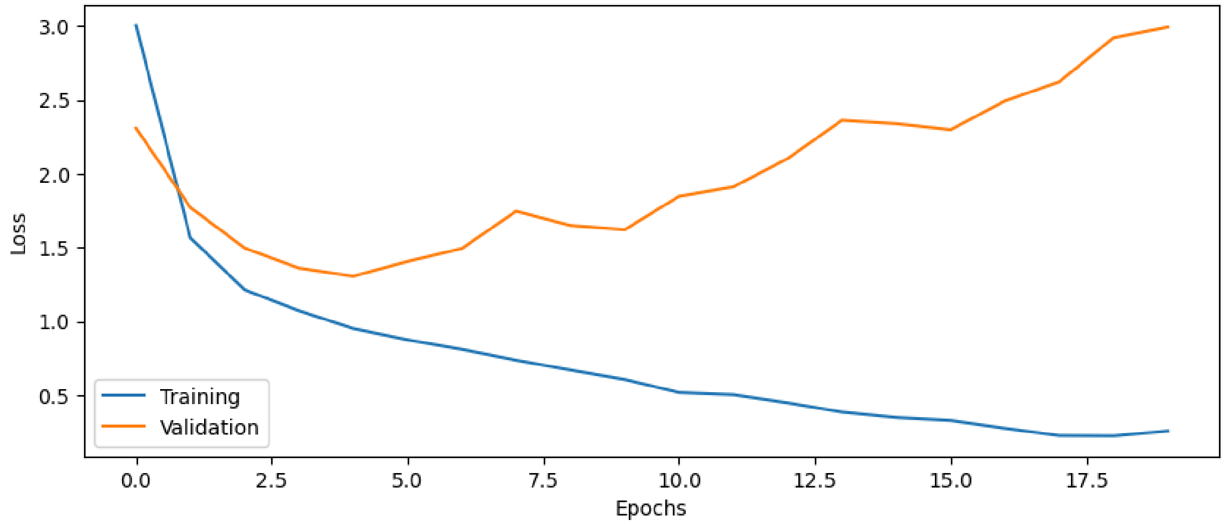
In [11]:
```
# Setup some training parameters
batch_size = 100
epochs = 20
input_shape = Xtrain.shape[1:]

# Build model
model1 = build_CNN(input_shape,n_conv_layers=2,n_dense_layers=0)

# Train the model  using training data and validation data
history1 = model1.fit(Xtrain,Ytrain,epochs=epochs, batch_size=batch_size,validation_da
```

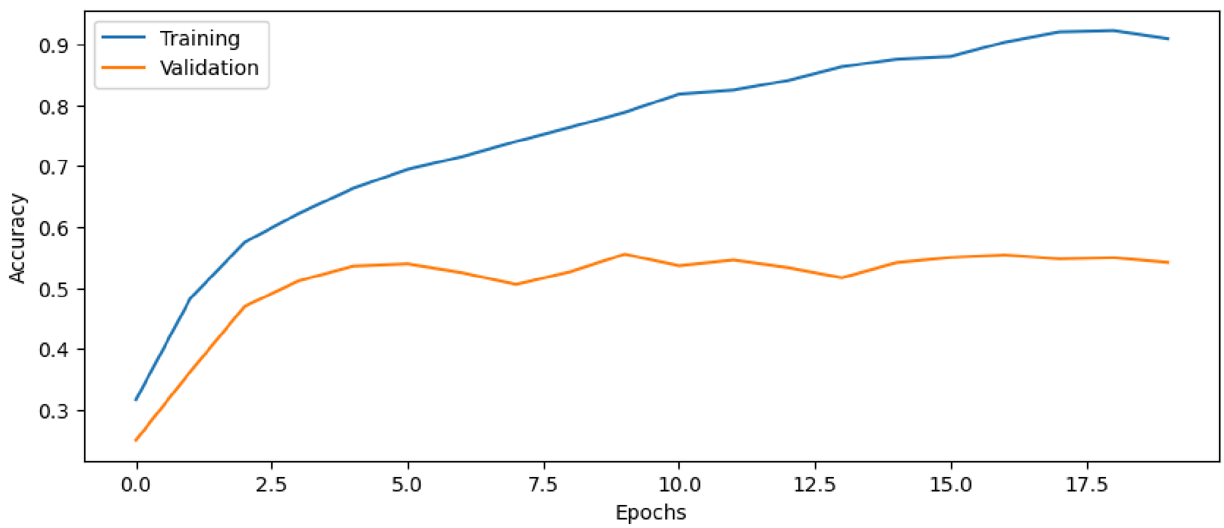
```
Epoch 1/20
70/70 [==============================] - 7s 43ms/step - loss: 3.0053 - accuracy: 0.31
69 - val_loss: 2.3078 - val_accuracy: 0.2503
Epoch 2/20
70/70 [==============================] - 3s 37ms/step - loss: 1.5653 - accuracy: 0.48
29 - val_loss: 1.7711 - val_accuracy: 0.3627
Epoch 3/20
70/70 [==============================] - 3s 37ms/step - loss: 1.2125 - accuracy: 0.57
51 - val_loss: 1.4955 - val_accuracy: 0.4700
Epoch 4/20
70/70 [==============================] - 3s 41ms/step - loss: 1.0707 - accuracy: 0.62
27 - val_loss: 1.3595 - val_accuracy: 0.5120
Epoch 5/20
70/70 [==============================] - 4s 52ms/step - loss: 0.9503 - accuracy: 0.66
39 - val_loss: 1.3043 - val_accuracy: 0.5360
Epoch 6/20
70/70 [==============================] - 3s 42ms/step - loss: 0.8733 - accuracy: 0.69
50 - val_loss: 1.4049 - val_accuracy: 0.5397
Epoch 7/20
70/70 [==============================] - 3s 42ms/step - loss: 0.8090 - accuracy: 0.71
57 - val_loss: 1.4924 - val_accuracy: 0.5253
Epoch 8/20
70/70 [==============================] - 3s 42ms/step - loss: 0.7354 - accuracy: 0.74
07 - val_loss: 1.7452 - val_accuracy: 0.5060
Epoch 9/20
70/70 [==============================] - 3s 42ms/step - loss: 0.6688 - accuracy: 0.76
41 - val_loss: 1.6483 - val_accuracy: 0.5270
Epoch 10/20
70/70 [==============================] - 3s 43ms/step - loss: 0.6048 - accuracy: 0.78
86 - val_loss: 1.6199 - val_accuracy: 0.5557
Epoch 11/20
70/70 [==============================] - 3s 40ms/step - loss: 0.5164 - accuracy: 0.81
86 - val_loss: 1.8463 - val_accuracy: 0.5367
Epoch 12/20
70/70 [==============================] - 3s 43ms/step - loss: 0.5015 - accuracy: 0.82
51 - val_loss: 1.9102 - val_accuracy: 0.5460
Epoch 13/20
70/70 [==============================] - 3s 42ms/step - loss: 0.4462 - accuracy: 0.84
09 - val_loss: 2.1020 - val_accuracy: 0.5337
Epoch 14/20
70/70 [==============================] - 3s 43ms/step - loss: 0.3837 - accuracy: 0.86
34 - val_loss: 2.3614 - val_accuracy: 0.5170
Epoch 15/20
70/70 [==============================] - 3s 44ms/step - loss: 0.3470 - accuracy: 0.87
60 - val_loss: 2.3386 - val_accuracy: 0.5417
Epoch 16/20
70/70 [==============================] - 3s 42ms/step - loss: 0.3271 - accuracy: 0.88
04 - val_loss: 2.2974 - val_accuracy: 0.5503
Epoch 17/20
70/70 [==============================] - 3s 41ms/step - loss: 0.2737 - accuracy: 0.90
39 - val_loss: 2.4923 - val_accuracy: 0.5543
Epoch 18/20
70/70 [==============================] - 3s 44ms/step - loss: 0.2246 - accuracy: 0.92
06 - val_loss: 2.6221 - val_accuracy: 0.5480
Epoch 19/20
70/70 [==============================] - 3s 43ms/step - loss: 0.2228 - accuracy: 0.92
27 - val_loss: 2.9206 - val_accuracy: 0.5500
Epoch 20/20
70/70 [==============================] - 3s 43ms/step - loss: 0.2554 - accuracy: 0.90
97 - val_loss: 2.9924 - val_accuracy: 0.5420
```

```
In [12]: # Evaluate the trained model on test set, not used in training or validation
score = model1.evaluate(Xtest,Ytest, batch_size=batch_size)
print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])
```

```
20/20 [==============================] - 0s 12ms/step - loss: 3.0391 - accuracy: 0.53
85
Test loss: 3.0391
Test accuracy: 0.5385
```

```
In [13]: # 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%) ?

- Test Accuracy- 0.5385
- Better than random chance, but still far away from satisfing.

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

- Training Accuracy- 0.9097
- Difference between training and test accuracy- 0.3712

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?

- A convolution layer will give n_filters output channels for each input image, where n_filters represents the number of filters, which will increase amount of input for the next convolution layer, which in turn will further multiply the input according to its number of filters. So it will be appropriate to use a smaller batch size to prevent out of memory errors.

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

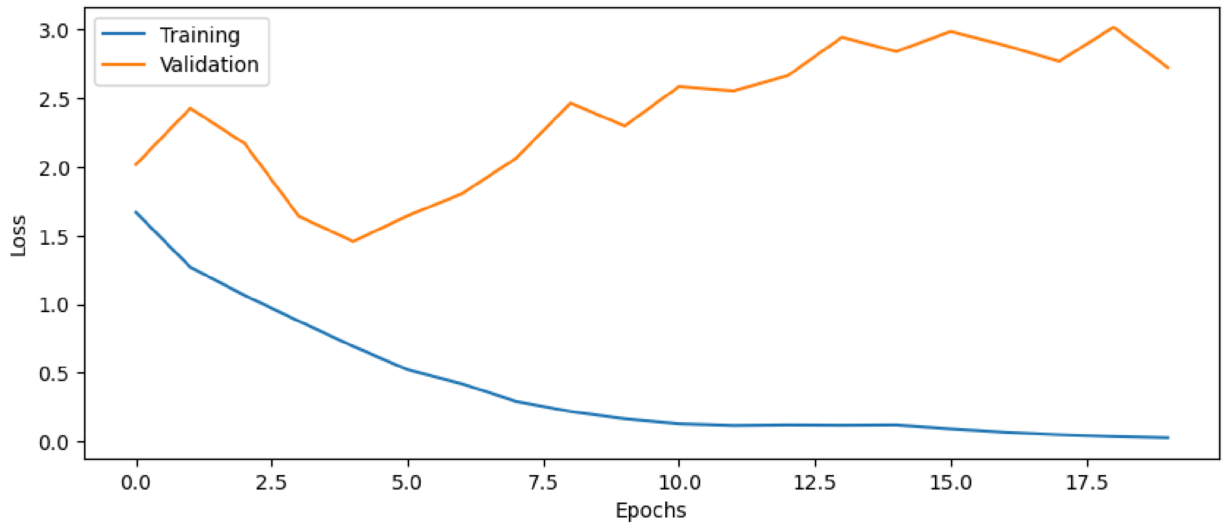
In [14]:
```
# Setup some training parameters
batch_size = 100
epochs = 20
input_shape = Xtrain.shape[1:]

# Build model
model2 = build_CNN(input_shape,n_conv_layers=2,n_dense_layers=1,n_nodes=50)

# Train the model  using training data and validation data
history2 = model2.fit(Xtrain,Ytrain,epochs=epochs, batch_size=batch_size,validation_da
```

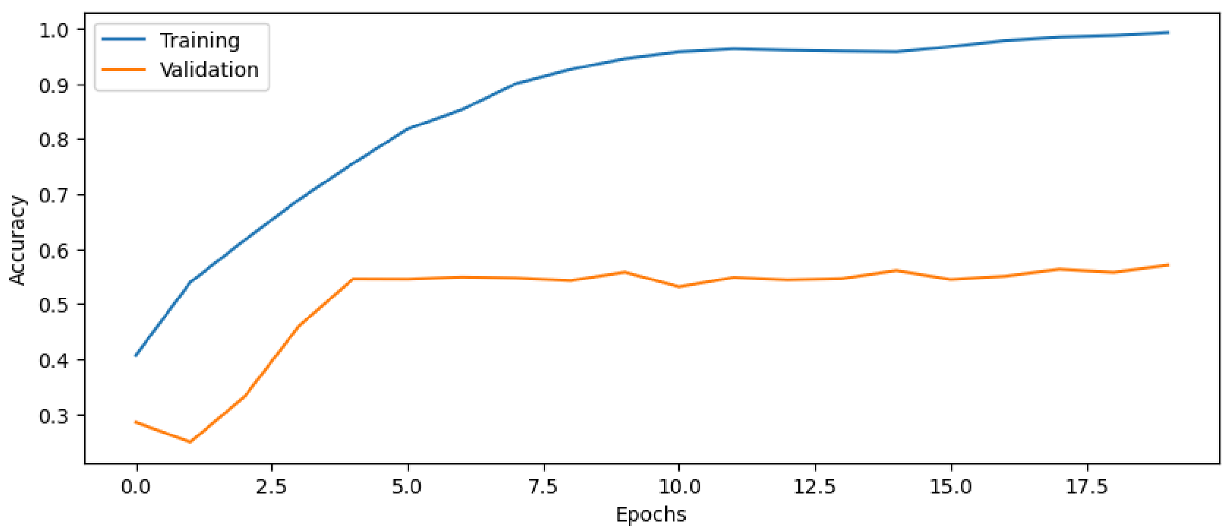
```
Epoch 1/20
70/70 [==============================] - 9s 77ms/step - loss: 1.6687 - accuracy: 0.40
77 - val_loss: 2.0188 - val_accuracy: 0.2857
Epoch 2/20
70/70 [==============================] - 3s 46ms/step - loss: 1.2701 - accuracy: 0.53
99 - val_loss: 2.4286 - val_accuracy: 0.2497
Epoch 3/20
70/70 [==============================] - 3s 45ms/step - loss: 1.0639 - accuracy: 0.61
64 - val_loss: 2.1713 - val_accuracy: 0.3323
Epoch 4/20
70/70 [==============================] - 4s 54ms/step - loss: 0.8760 - accuracy: 0.69
00 - val_loss: 1.6404 - val_accuracy: 0.4603
Epoch 5/20
70/70 [==============================] - 4s 59ms/step - loss: 0.6923 - accuracy: 0.75
59 - val_loss: 1.4571 - val_accuracy: 0.5460
Epoch 6/20
70/70 [==============================] - 3s 43ms/step - loss: 0.5223 - accuracy: 0.81
83 - val_loss: 1.6441 - val_accuracy: 0.5457
Epoch 7/20
70/70 [==============================] - 4s 57ms/step - loss: 0.4198 - accuracy: 0.85
33 - val_loss: 1.8051 - val_accuracy: 0.5493
Epoch 8/20
70/70 [==============================] - 3s 47ms/step - loss: 0.2907 - accuracy: 0.90
07 - val_loss: 2.0637 - val_accuracy: 0.5477
Epoch 9/20
70/70 [==============================] - 3s 47ms/step - loss: 0.2175 - accuracy: 0.92
64 - val_loss: 2.4660 - val_accuracy: 0.5430
Epoch 10/20
70/70 [==============================] - 4s 62ms/step - loss: 0.1643 - accuracy: 0.94
59 - val_loss: 2.2993 - val_accuracy: 0.5583
Epoch 11/20
70/70 [==============================] - 4s 60ms/step - loss: 0.1280 - accuracy: 0.95
83 - val_loss: 2.5872 - val_accuracy: 0.5317
Epoch 12/20
70/70 [==============================] - 4s 55ms/step - loss: 0.1157 - accuracy: 0.96
39 - val_loss: 2.5553 - val_accuracy: 0.5487
Epoch 13/20
70/70 [==============================] - 4s 52ms/step - loss: 0.1194 - accuracy: 0.96
14 - val_loss: 2.6657 - val_accuracy: 0.5443
Epoch 14/20
70/70 [==============================] - 4s 53ms/step - loss: 0.1175 - accuracy: 0.95
97 - val_loss: 2.9456 - val_accuracy: 0.5467
Epoch 15/20
70/70 [==============================] - 3s 45ms/step - loss: 0.1195 - accuracy: 0.95
84 - val_loss: 2.8421 - val_accuracy: 0.5613
Epoch 16/20
70/70 [==============================] - 3s 46ms/step - loss: 0.0913 - accuracy: 0.96
79 - val_loss: 2.9875 - val_accuracy: 0.5450
Epoch 17/20
70/70 [==============================] - 3s 46ms/step - loss: 0.0666 - accuracy: 0.97
89 - val_loss: 2.8869 - val_accuracy: 0.5510
Epoch 18/20
70/70 [==============================] - 3s 49ms/step - loss: 0.0480 - accuracy: 0.98
51 - val_loss: 2.7709 - val_accuracy: 0.5637
Epoch 19/20
70/70 [==============================] - 4s 51ms/step - loss: 0.0350 - accuracy: 0.98
80 - val_loss: 3.0187 - val_accuracy: 0.5580
Epoch 20/20
70/70 [==============================] - 3s 44ms/step - loss: 0.0255 - accuracy: 0.99
31 - val_loss: 2.7259 - val_accuracy: 0.5710
```

In [15]:
```
# Evaluate the trained model on test set, not used in training or validation
score = model2.evaluate(Xtest,Ytest, batch_size=batch_size)
print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])
```

```
20/20 [==============================] - 0s 16ms/step - loss: 2.7891 - accuracy: 0.56
30
Test loss: 2.7891
Test accuracy: 0.5630
```

In [16]:
```
# Plot the history from the training run
plot_results(history2)
```





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

In [27]:
```
# Setup some training parameters
batch_size = 100
epochs = 20
input_shape = Xtrain.shape[1:]

# Build model
model3 = build_CNN(input_shape,n_conv_layers=4,n_dense_layers=1,n_nodes=50)
```

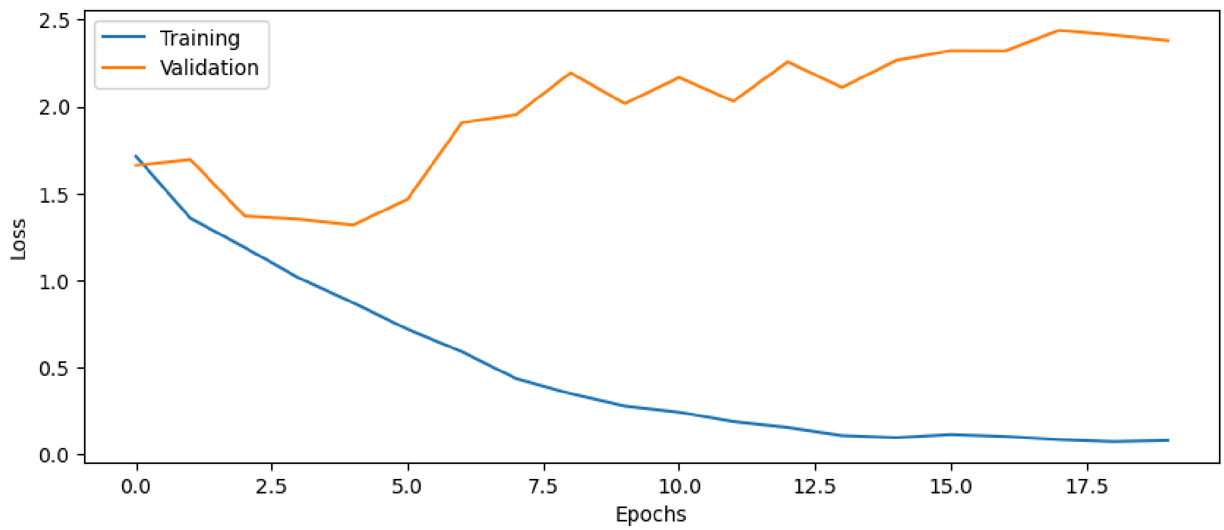
```
# Train the model  using training data and validation data
history3 = model3.fit(Xtrain,Ytrain,epochs=epochs, batch_size=batch_size,validation_da
```

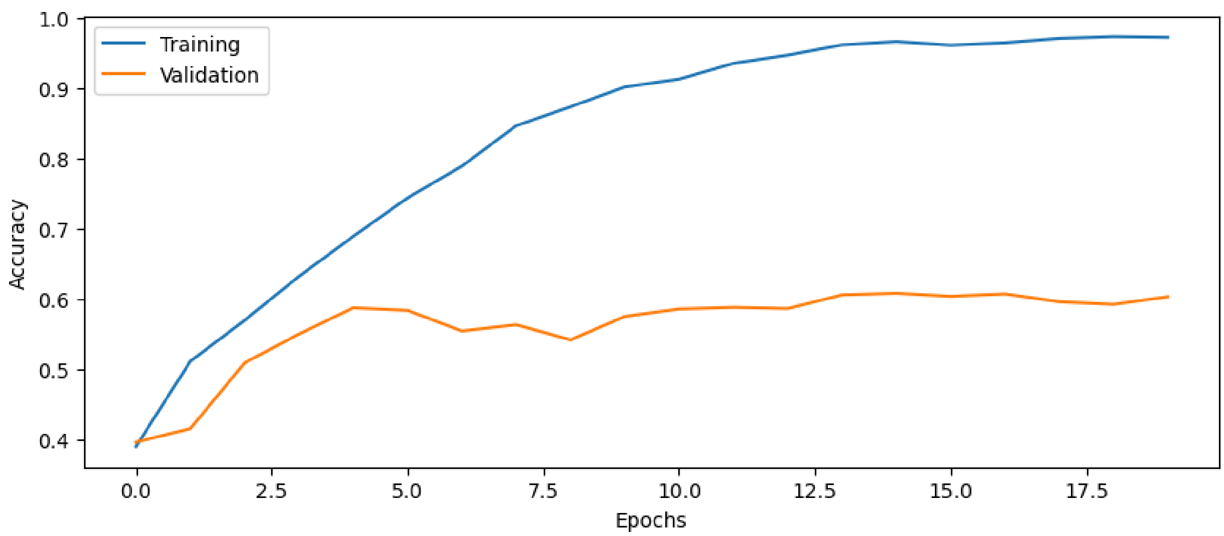
```
Epoch 1/20
70/70 [==============================] - 6s 65ms/step - loss: 1.7139 - accuracy: 0.39
00 - val_loss: 1.6617 - val_accuracy: 0.3963
Epoch 2/20
70/70 [==============================] - 4s 62ms/step - loss: 1.3583 - accuracy: 0.51
11 - val_loss: 1.6959 - val_accuracy: 0.4153
Epoch 3/20
70/70 [==============================] - 5s 65ms/step - loss: 1.1894 - accuracy: 0.56
96 - val_loss: 1.3700 - val_accuracy: 0.5090
Epoch 4/20
70/70 [==============================] - 4s 64ms/step - loss: 1.0139 - accuracy: 0.63
20 - val_loss: 1.3519 - val_accuracy: 0.5500
Epoch 5/20
70/70 [==============================] - 4s 63ms/step - loss: 0.8714 - accuracy: 0.68
90 - val_loss: 1.3188 - val_accuracy: 0.5873
Epoch 6/20
70/70 [==============================] - 4s 64ms/step - loss: 0.7189 - accuracy: 0.74
37 - val_loss: 1.4669 - val_accuracy: 0.5837
Epoch 7/20
70/70 [==============================] - 4s 64ms/step - loss: 0.5899 - accuracy: 0.78
89 - val_loss: 1.9075 - val_accuracy: 0.5543
Epoch 8/20
70/70 [==============================] - 5s 65ms/step - loss: 0.4351 - accuracy: 0.84
64 - val_loss: 1.9538 - val_accuracy: 0.5633
Epoch 9/20
70/70 [==============================] - 4s 62ms/step - loss: 0.3487 - accuracy: 0.87
34 - val_loss: 2.1939 - val_accuracy: 0.5417
Epoch 10/20
70/70 [==============================] - 5s 68ms/step - loss: 0.2773 - accuracy: 0.90
16 - val_loss: 2.0173 - val_accuracy: 0.5750
Epoch 11/20
70/70 [==============================] - 5s 71ms/step - loss: 0.2420 - accuracy: 0.91
27 - val_loss: 2.1671 - val_accuracy: 0.5857
Epoch 12/20
70/70 [==============================] - 5s 63ms/step - loss: 0.1876 - accuracy: 0.93
51 - val_loss: 2.0306 - val_accuracy: 0.5880
Epoch 13/20
70/70 [==============================] - 5s 65ms/step - loss: 0.1542 - accuracy: 0.94
69 - val_loss: 2.2573 - val_accuracy: 0.5863
Epoch 14/20
70/70 [==============================] - 5s 65ms/step - loss: 0.1079 - accuracy: 0.96
14 - val_loss: 2.1098 - val_accuracy: 0.6057
Epoch 15/20
70/70 [==============================] - 5s 64ms/step - loss: 0.0955 - accuracy: 0.96
61 - val_loss: 2.2658 - val_accuracy: 0.6080
Epoch 16/20
70/70 [==============================] - 5s 67ms/step - loss: 0.1140 - accuracy: 0.96
10 - val_loss: 2.3211 - val_accuracy: 0.6037
Epoch 17/20
70/70 [==============================] - 5s 65ms/step - loss: 0.1026 - accuracy: 0.96
44 - val_loss: 2.3198 - val_accuracy: 0.6070
Epoch 18/20
70/70 [==============================] - 5s 65ms/step - loss: 0.0841 - accuracy: 0.97
07 - val_loss: 2.4384 - val_accuracy: 0.5960
Epoch 19/20
70/70 [==============================] - 4s 63ms/step - loss: 0.0724 - accuracy: 0.97
33 - val_loss: 2.4108 - val_accuracy: 0.5923
Epoch 20/20
70/70 [==============================] - 5s 65ms/step - loss: 0.0792 - accuracy: 0.97
23 - val_loss: 2.3797 - val_accuracy: 0.6030
```

In [28]:
```
# Evaluate the trained model on test set, not used in training or validation
score = model3.evaluate(Xtest,Ytest, batch_size=batch_size)
print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])
```

```
20/20 [==============================] - 0s 16ms/step - loss: 2.4584 - accuracy: 0.58
25
Test loss: 2.4584
Test accuracy: 0.5825
```

In [29]:
```
# 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?

- Trainable params: 124,180
- The last convolution layer has most parameters.

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

- The input is the output of last (max_pooling2d) layer. The output is the same image dimension as input but with a doubled channels for each pixel

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

- Yes, 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?

- It is equal to 128

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)?

- The number of parameters for each convolution layer is equal to: n_filters *n_coefficients* n_input_channels + biases. This is because it has separate coefficients for each input channel.

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

- MaxPooling layer reduces the size of output channels of the previous convolution layer, so the output channel size will progressively decrease after each pooling layer. Although it does not impact the number of parameters for the next convolution layer, it does have a great effect on the output size of the flatten layer, which reduces the number of parameters for the subsequent dense layers.

In [30]:

```
# Print network architecture

model3.summary()
```

```
Model: "sequential_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 conv2d_16 (Conv2D)          (None, 32, 32, 16)        448       
                                                                 
 batch_normalization_20 (Bat  (None, 32, 32, 16)       64        
 chNormalization)                                                
                                                                 
 max_pooling2d_16 (MaxPoolin  (None, 16, 16, 16)       0         
 g2D)                                                            
                                                                 
 conv2d_17 (Conv2D)          (None, 16, 16, 32)        4640      
                                                                 
 batch_normalization_21 (Bat  (None, 16, 16, 32)       128       
 chNormalization)                                                
                                                                 
 max_pooling2d_17 (MaxPoolin  (None, 8, 8, 32)         0         
 g2D)                                                            
                                                                 
 conv2d_18 (Conv2D)          (None, 8, 8, 64)          18496     
                                                                 
 batch_normalization_22 (Bat  (None, 8, 8, 64)         256       
 chNormalization)                                                
                                                                 
 max_pooling2d_18 (MaxPoolin  (None, 4, 4, 64)         0         
 g2D)                                                            
                                                                 
 conv2d_19 (Conv2D)          (None, 4, 4, 128)         73856     
                                                                 
 batch_normalization_23 (Bat  (None, 4, 4, 128)        512       
 chNormalization)                                                
                                                                 
 max_pooling2d_19 (MaxPoolin  (None, 2, 2, 128)        0         
 g2D)                                                            
                                                                 
 flatten_5 (Flatten)         (None, 512)               0         
                                                                 
 dense_9 (Dense)             (None, 50)                25650     
                                                                 
 batch_normalization_24 (Bat  (None, 50)               200       
 chNormalization)                                                
                                                                 
 dense_10 (Dense)            (None, 10)                510       
                                                                 
=================================================================
Total params: 124,760
Trainable params: 124,180
Non-trainable params: 580
_________________________________________________________________
```

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?

- Improvment- 0.6200- 0.5825= 0.0375

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

- L1/L2 regularization, we can add augment kernel_regularizer/bias_regularizer/activity_regularizer = 'l2' to conv2D

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

In [31]:

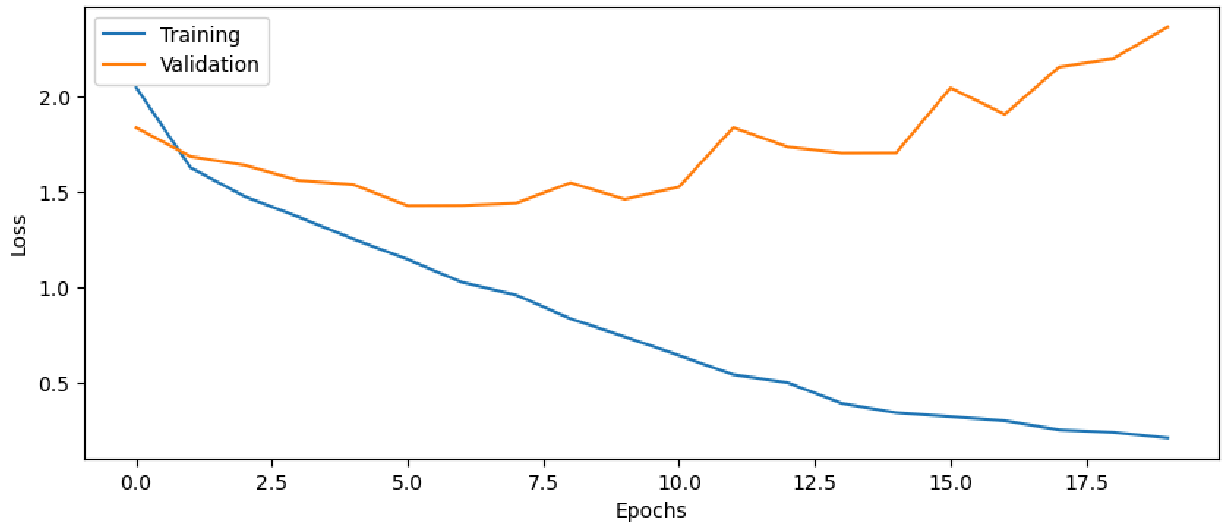
```
# Setup some training parameters
batch_size = 100
epochs = 20
input_shape = Xtrain.shape[1:]

# Build model
model4 = build_CNN(input_shape,n_conv_layers=4,n_dense_layers=1,n_nodes=50, use_dropou

# Train the model  using training data and validation data
history4 = model4.fit(Xtrain,Ytrain,epochs=epochs, batch_size=batch_size,validation_da
```

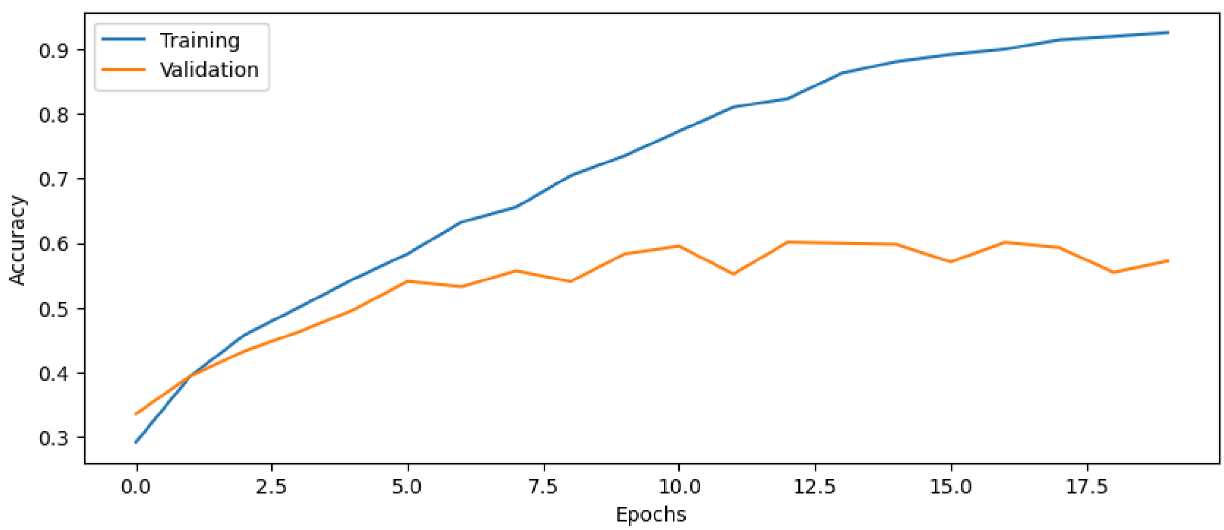
```
Epoch 1/20
70/70 [==============================] - 8s 89ms/step - loss: 2.0586 - accuracy: 0.27
97 - val_loss: 2.7289 - val_accuracy: 0.2180
Epoch 2/20
70/70 [==============================] - 5s 68ms/step - loss: 1.6305 - accuracy: 0.38
63 - val_loss: 1.7923 - val_accuracy: 0.3440
Epoch 3/20
70/70 [==============================] - 5s 77ms/step - loss: 1.4926 - accuracy: 0.44
20 - val_loss: 1.7760 - val_accuracy: 0.3730
Epoch 4/20
70/70 [==============================] - 5s 74ms/step - loss: 1.3768 - accuracy: 0.49
54 - val_loss: 1.7726 - val_accuracy: 0.3990
Epoch 5/20
70/70 [==============================] - 5s 67ms/step - loss: 1.2615 - accuracy: 0.53
80 - val_loss: 1.4094 - val_accuracy: 0.5023
Epoch 6/20
70/70 [==============================] - 5s 68ms/step - loss: 1.1577 - accuracy: 0.58
44 - val_loss: 1.5373 - val_accuracy: 0.5087
Epoch 7/20
70/70 [==============================] - 4s 62ms/step - loss: 1.0661 - accuracy: 0.61
26 - val_loss: 1.3815 - val_accuracy: 0.5317
Epoch 8/20
70/70 [==============================] - 4s 62ms/step - loss: 0.9724 - accuracy: 0.65
60 - val_loss: 1.5391 - val_accuracy: 0.5247
Epoch 9/20
70/70 [==============================] - 5s 66ms/step - loss: 0.8933 - accuracy: 0.68
71 - val_loss: 1.4325 - val_accuracy: 0.5350
Epoch 10/20
70/70 [==============================] - 5s 67ms/step - loss: 0.7976 - accuracy: 0.71
87 - val_loss: 1.5487 - val_accuracy: 0.5563
Epoch 11/20
70/70 [==============================] - 4s 60ms/step - loss: 0.7027 - accuracy: 0.75
09 - val_loss: 1.7527 - val_accuracy: 0.5623
Epoch 12/20
70/70 [==============================] - 4s 64ms/step - loss: 0.6163 - accuracy: 0.78
27 - val_loss: 1.7610 - val_accuracy: 0.5433
Epoch 13/20
70/70 [==============================] - 4s 60ms/step - loss: 0.5367 - accuracy: 0.80
86 - val_loss: 1.8082 - val_accuracy: 0.5657
Epoch 14/20
70/70 [==============================] - 4s 64ms/step - loss: 0.4798 - accuracy: 0.83
13 - val_loss: 1.6669 - val_accuracy: 0.5877
Epoch 15/20
70/70 [==============================] - 4s 62ms/step - loss: 0.4107 - accuracy: 0.85
80 - val_loss: 2.1823 - val_accuracy: 0.5623
Epoch 16/20
70/70 [==============================] - 4s 63ms/step - loss: 0.3332 - accuracy: 0.88
27 - val_loss: 2.0943 - val_accuracy: 0.5720
Epoch 17/20
70/70 [==============================] - 5s 66ms/step - loss: 0.3058 - accuracy: 0.89
43 - val_loss: 1.8763 - val_accuracy: 0.6007
Epoch 18/20
70/70 [==============================] - 4s 62ms/step - loss: 0.2969 - accuracy: 0.90
06 - val_loss: 2.0780 - val_accuracy: 0.5843
Epoch 19/20
70/70 [==============================] - 4s 64ms/step - loss: 0.2502 - accuracy: 0.91
33 - val_loss: 2.0895 - val_accuracy: 0.5990
Epoch 20/20
70/70 [==============================] - 4s 62ms/step - loss: 0.2113 - accuracy: 0.92
87 - val_loss: 1.9192 - val_accuracy: 0.6170
```

In [32]:
```
# Evaluate the trained model on test set, not used in training or validation
score = model4.evaluate(Xtest,Ytest, batch_size=batch_size)
print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])
```

```
20/20 [==============================] - 0s 15ms/step - loss: 1.9620 - accuracy: 0.62
00
Test loss: 1.9620
Test accuracy: 0.6200
```

In [26]:
```
# 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?

Your best config

In [36]:
```
# Setup some training parameters
batch_size = 10
epochs = 30
input_shape = Xtrain.shape[1:]

# Build model
model5 = build_CNN(input_shape,n_conv_layers=4,n_dense_layers=6,n_nodes=200, use_dropo

# Train the model  using training data and validation data
history5 = model5.fit(Xtrain,Ytrain,epochs=epochs, batch_size=batch_size,validation_da
```

```
Epoch 1/30
700/700 [==============================] - 19s 22ms/step - loss: 2.4213 - accuracy:
0.1770 - val_loss: 1.9721 - val_accuracy: 0.2363
Epoch 2/30
700/700 [==============================] - 16s 23ms/step - loss: 2.0174 - accuracy:
0.2584 - val_loss: 1.7900 - val_accuracy: 0.3313
Epoch 3/30
700/700 [==============================] - 16s 23ms/step - loss: 1.9026 - accuracy:
0.2989 - val_loss: 1.7814 - val_accuracy: 0.3230
Epoch 4/30
700/700 [==============================] - 17s 24ms/step - loss: 1.8452 - accuracy:
0.3201 - val_loss: 2.1085 - val_accuracy: 0.2843
Epoch 5/30
700/700 [==============================] - 17s 24ms/step - loss: 1.8045 - accuracy:
0.3286 - val_loss: 1.5732 - val_accuracy: 0.4057
Epoch 6/30
700/700 [==============================] - 16s 23ms/step - loss: 1.7584 - accuracy:
0.3446 - val_loss: 1.5802 - val_accuracy: 0.4067
Epoch 7/30
700/700 [==============================] - 17s 24ms/step - loss: 1.7212 - accuracy:
0.3600 - val_loss: 1.6029 - val_accuracy: 0.3827
Epoch 8/30
700/700 [==============================] - 16s 23ms/step - loss: 1.6851 - accuracy:
0.3824 - val_loss: 1.7327 - val_accuracy: 0.3703
Epoch 9/30
700/700 [==============================] - 17s 25ms/step - loss: 1.6678 - accuracy:
0.3837 - val_loss: 1.6826 - val_accuracy: 0.3857
Epoch 10/30
700/700 [==============================] - 16s 24ms/step - loss: 1.6004 - accuracy:
0.4109 - val_loss: 1.4777 - val_accuracy: 0.4393
Epoch 11/30
700/700 [==============================] - 17s 24ms/step - loss: 1.6111 - accuracy:
0.4074 - val_loss: 1.3792 - val_accuracy: 0.4697
Epoch 12/30
700/700 [==============================] - 17s 24ms/step - loss: 1.5242 - accuracy:
0.4466 - val_loss: 1.3691 - val_accuracy: 0.5003
Epoch 13/30
700/700 [==============================] - 17s 24ms/step - loss: 1.5591 - accuracy:
0.4399 - val_loss: 1.4752 - val_accuracy: 0.4550
Epoch 14/30
700/700 [==============================] - 17s 24ms/step - loss: 1.5120 - accuracy:
0.4616 - val_loss: 1.3375 - val_accuracy: 0.5053
Epoch 15/30
700/700 [==============================] - 17s 24ms/step - loss: 1.4418 - accuracy:
0.4834 - val_loss: 1.3423 - val_accuracy: 0.5150
Epoch 16/30
700/700 [==============================] - 17s 24ms/step - loss: 1.3948 - accuracy:
0.5074 - val_loss: 1.3406 - val_accuracy: 0.5307
Epoch 17/30
700/700 [==============================] - 17s 24ms/step - loss: 1.3753 - accuracy:
0.5193 - val_loss: 1.2333 - val_accuracy: 0.5640
Epoch 18/30
700/700 [==============================] - 17s 24ms/step - loss: 1.3126 - accuracy:
0.5386 - val_loss: 1.2367 - val_accuracy: 0.5597
Epoch 19/30
700/700 [==============================] - 19s 27ms/step - loss: 1.2399 - accuracy:
0.5644 - val_loss: 1.3508 - val_accuracy: 0.5357
Epoch 20/30
700/700 [==============================] - 17s 25ms/step - loss: 1.2384 - accuracy:
0.5727 - val_loss: 1.1895 - val_accuracy: 0.5767
```

```
Epoch 21/30
700/700 [==============================] - 17s 24ms/step - loss: 1.2093 - accuracy:
0.5816 - val_loss: 1.3312 - val_accuracy: 0.5263
Epoch 22/30
700/700 [==============================] - 16s 24ms/step - loss: 1.1420 - accuracy:
0.6103 - val_loss: 1.3563 - val_accuracy: 0.5540
Epoch 23/30
700/700 [==============================] - 17s 24ms/step - loss: 1.1008 - accuracy:
0.6263 - val_loss: 1.2162 - val_accuracy: 0.5803
Epoch 24/30
700/700 [==============================] - 16s 23ms/step - loss: 1.0820 - accuracy:
0.6261 - val_loss: 1.2571 - val_accuracy: 0.5760
Epoch 25/30
700/700 [==============================] - 16s 23ms/step - loss: 1.0639 - accuracy:
0.6386 - val_loss: 1.2998 - val_accuracy: 0.5633
Epoch 26/30
700/700 [==============================] - 16s 24ms/step - loss: 1.0125 - accuracy:
0.6527 - val_loss: 1.2377 - val_accuracy: 0.5773
Epoch 27/30
700/700 [==============================] - 17s 24ms/step - loss: 1.0550 - accuracy:
0.6374 - val_loss: 1.2279 - val_accuracy: 0.5943
Epoch 28/30
700/700 [==============================] - 17s 24ms/step - loss: 0.9521 - accuracy:
0.6716 - val_loss: 1.2351 - val_accuracy: 0.5967
Epoch 29/30
700/700 [==============================] - 17s 25ms/step - loss: 0.9950 - accuracy:
0.6640 - val_loss: 1.2078 - val_accuracy: 0.5950
Epoch 30/30
700/700 [==============================] - 17s 24ms/step - loss: 0.9329 - accuracy:
0.6926 - val_loss: 1.2289 - val_accuracy: 0.5883
```

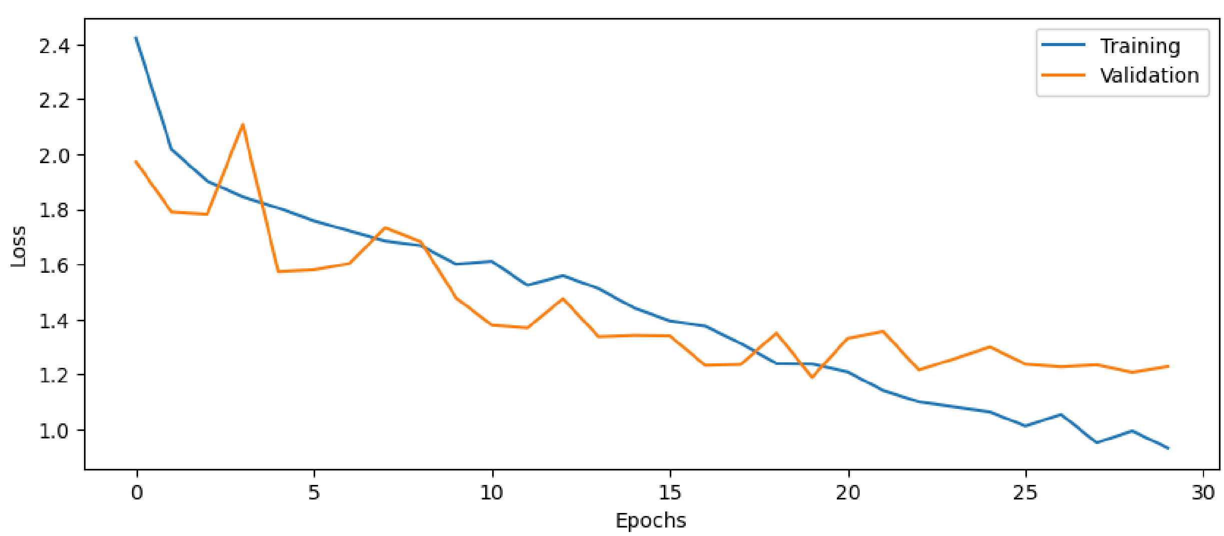
In [37]:

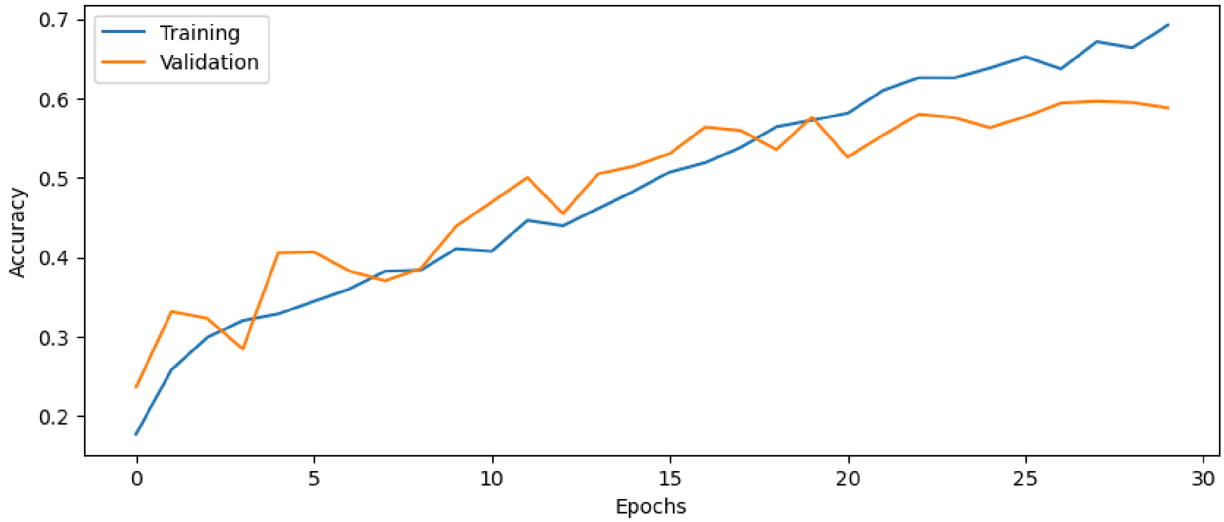
```python
# Evaluate the trained model on test set, not used in training or validation
score = model5.evaluate(Xtest,Ytest, batch_size=batch_size)
print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])
```

```
200/200 [==============================] - 1s 6ms/step - loss: 1.2656 - accuracy: 0.5
670
Test loss: 1.2656
Test accuracy: 0.5670
```

In [38]:

```python
# Plot the history from the training run
plot_results(history5)
```





Part 16: Rotate the test images

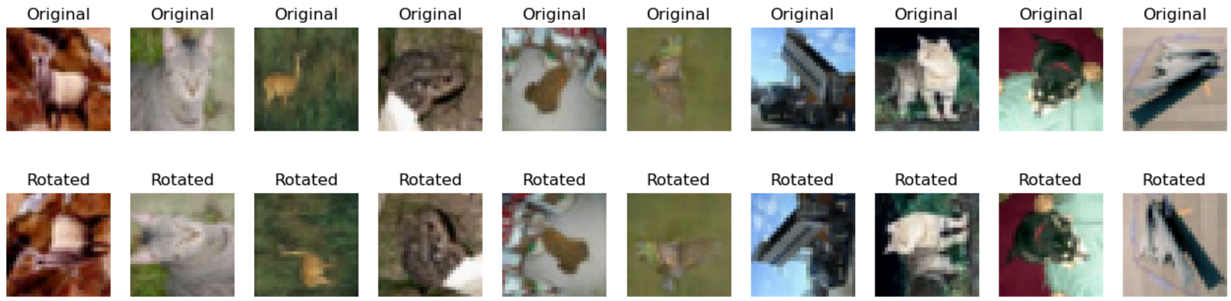
How high is the test accuracy if we rotate the test images? In other words, how good is the CNN at generalizing to rotated images?

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.

In [20]:
```
def myrotate(images):

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

    return images_rot
```

In [21]:
```
# 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 [41]:
```
# Evaluate the trained model on rotated test set
score = model5.evaluate(Xtest_rotated,Ytest, batch_size=batch_size)
print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])
```

```
200/200 [==============================] - 1s 6ms/step - loss: 3.3022 - accuracy: 0.2
135
Test loss: 3.3022
Test accuracy: 0.2135
```

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 [14]:
```
# 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 [15]:

```
# Set up a data generator with on-the-fly data augmentation, 20% validation split
# Use a rotation range of 30 degrees, horizontal and vertical flipping
from keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(rotation_range=0.3,horizontal_flip=True, vertical_flip=Tr

datagen.fit(Xtrain)

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

# Setup a flow for validation data, assume that we can fit all images into CPU memory
datagen.flow(x=Xtrain,y=Ytrain, subset='validation')
```

Out[15]:

```
<keras.preprocessing.image.NumpyArrayIterator at 0x1a908667b20>
```

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?

In [16]:

```
# 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?

- When using data augmentation during training, the training accuracy often increases more slowly compared to training without augmentation. The reversed/rotated images(but with same label) put an extra obstacle towards learning, which makes the model takes loger time to learn

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

- Transformations such as color augmentation, flips, shifts, and scaling to the original images.

In [18]:
```
# Setup some training parameters
batch_size = 100
epochs = 200
input_shape = Xtrain.shape[1:]

# Build model (your best config)
model6 = build_CNN(input_shape,n_conv_layers=4,n_dense_layers=6,n_nodes=200, use_dropc

validation_split=0.2

# Train the model using on the fly augmentation
history6 = model6.fit(datagen.flow(x=Xtrain,y=Ytrain, subset='training',batch_size=bat
                      validation_data = datagen.flow(x=Xtrain,y=Ytrain, subset='valida
                      steps_per_epoch = len(Xtrain)*(1 - validation_split)/batch_size,
                      validation_steps = len(Xtrain)*validation_split/batch_size,\
                      epochs=epochs)
```

```
Epoch 1/200
80/80 [==============================] - 17s 162ms/step - loss: 2.6003 - accuracy: 0.
1523 - val_loss: 5.1737 - val_accuracy: 0.1390
Epoch 2/200
80/80 [==============================] - 15s 188ms/step - loss: 2.0277 - accuracy: 0.
2391 - val_loss: 3.4374 - val_accuracy: 0.1570
Epoch 3/200
80/80 [==============================] - 15s 181ms/step - loss: 1.8825 - accuracy: 0.
2665 - val_loss: 2.8377 - val_accuracy: 0.1700
Epoch 4/200
80/80 [==============================] - 15s 183ms/step - loss: 1.7979 - accuracy: 0.
3064 - val_loss: 2.4330 - val_accuracy: 0.1970
Epoch 5/200
80/80 [==============================] - 15s 182ms/step - loss: 1.7133 - accuracy: 0.
3470 - val_loss: 1.8015 - val_accuracy: 0.3380
Epoch 6/200
80/80 [==============================] - 15s 183ms/step - loss: 1.6464 - accuracy: 0.
3749 - val_loss: 1.8612 - val_accuracy: 0.3555
Epoch 7/200
80/80 [==============================] - 15s 183ms/step - loss: 1.5834 - accuracy: 0.
3994 - val_loss: 1.6341 - val_accuracy: 0.3940
Epoch 8/200
80/80 [==============================] - 15s 182ms/step - loss: 1.5282 - accuracy: 0.
4283 - val_loss: 1.6899 - val_accuracy: 0.4390
Epoch 9/200
80/80 [==============================] - 15s 182ms/step - loss: 1.4782 - accuracy: 0.
4507 - val_loss: 1.6111 - val_accuracy: 0.4525
Epoch 10/200
80/80 [==============================] - 15s 183ms/step - loss: 1.4484 - accuracy: 0.
4721 - val_loss: 1.6414 - val_accuracy: 0.4345
Epoch 11/200
80/80 [==============================] - 15s 183ms/step - loss: 1.3950 - accuracy: 0.
4963 - val_loss: 1.4520 - val_accuracy: 0.5000
Epoch 12/200
80/80 [==============================] - 15s 182ms/step - loss: 1.3655 - accuracy: 0.
5017 - val_loss: 1.4740 - val_accuracy: 0.5110
Epoch 13/200
80/80 [==============================] - 15s 183ms/step - loss: 1.3215 - accuracy: 0.
5249 - val_loss: 1.4999 - val_accuracy: 0.4765
Epoch 14/200
80/80 [==============================] - 15s 185ms/step - loss: 1.2908 - accuracy: 0.
5390 - val_loss: 1.4069 - val_accuracy: 0.5310
Epoch 15/200
80/80 [==============================] - 32s 405ms/step - loss: 1.2584 - accuracy: 0.
5460 - val_loss: 1.4422 - val_accuracy: 0.5030
Epoch 16/200
80/80 [==============================] - 30s 368ms/step - loss: 1.2576 - accuracy: 0.
5519 - val_loss: 1.4429 - val_accuracy: 0.5095
Epoch 17/200
80/80 [==============================] - 22s 278ms/step - loss: 1.2053 - accuracy: 0.
5696 - val_loss: 1.3946 - val_accuracy: 0.5345
Epoch 18/200
80/80 [==============================] - 22s 278ms/step - loss: 1.1923 - accuracy: 0.
5746 - val_loss: 1.7928 - val_accuracy: 0.4575
Epoch 19/200
80/80 [==============================] - 22s 279ms/step - loss: 1.1719 - accuracy: 0.
5872 - val_loss: 1.3768 - val_accuracy: 0.5585
Epoch 20/200
80/80 [==============================] - 20s 255ms/step - loss: 1.1224 - accuracy: 0.
6051 - val_loss: 1.2758 - val_accuracy: 0.5535
```

```
Epoch 21/200
80/80 [==============================] - 21s 258ms/step - loss: 1.1150 - accuracy: 0.
6144 - val_loss: 1.3171 - val_accuracy: 0.5580
Epoch 22/200
80/80 [==============================] - 22s 276ms/step - loss: 1.0917 - accuracy: 0.
6159 - val_loss: 1.4915 - val_accuracy: 0.5145
Epoch 23/200
80/80 [==============================] - 23s 293ms/step - loss: 1.0965 - accuracy: 0.
6187 - val_loss: 1.2066 - val_accuracy: 0.5905
Epoch 24/200
80/80 [==============================] - 21s 255ms/step - loss: 1.0639 - accuracy: 0.
6260 - val_loss: 1.3886 - val_accuracy: 0.5655
Epoch 25/200
80/80 [==============================] - 21s 261ms/step - loss: 1.0240 - accuracy: 0.
6414 - val_loss: 1.2084 - val_accuracy: 0.5930
Epoch 26/200
80/80 [==============================] - 22s 272ms/step - loss: 1.0264 - accuracy: 0.
6414 - val_loss: 1.2496 - val_accuracy: 0.5775
Epoch 27/200
80/80 [==============================] - 21s 259ms/step - loss: 1.0005 - accuracy: 0.
6504 - val_loss: 1.1947 - val_accuracy: 0.6010
Epoch 28/200
80/80 [==============================] - 22s 273ms/step - loss: 0.9780 - accuracy: 0.
6531 - val_loss: 1.2064 - val_accuracy: 0.5755
Epoch 29/200
80/80 [==============================] - 20s 247ms/step - loss: 0.9716 - accuracy: 0.
6659 - val_loss: 1.2323 - val_accuracy: 0.6065
Epoch 30/200
80/80 [==============================] - 22s 270ms/step - loss: 0.9660 - accuracy: 0.
6660 - val_loss: 1.3472 - val_accuracy: 0.5695
Epoch 31/200
80/80 [==============================] - 17s 212ms/step - loss: 0.9606 - accuracy: 0.
6680 - val_loss: 1.3599 - val_accuracy: 0.5705
Epoch 32/200
80/80 [==============================] - 17s 211ms/step - loss: 0.9008 - accuracy: 0.
6870 - val_loss: 1.2099 - val_accuracy: 0.5990
Epoch 33/200
80/80 [==============================] - 21s 266ms/step - loss: 0.9442 - accuracy: 0.
6746 - val_loss: 1.2350 - val_accuracy: 0.6110
Epoch 34/200
80/80 [==============================] - 19s 237ms/step - loss: 0.8865 - accuracy: 0.
6977 - val_loss: 1.1242 - val_accuracy: 0.6235
Epoch 35/200
80/80 [==============================] - 19s 236ms/step - loss: 0.8847 - accuracy: 0.
6950 - val_loss: 1.2954 - val_accuracy: 0.5985
Epoch 36/200
80/80 [==============================] - 17s 209ms/step - loss: 0.8769 - accuracy: 0.
6991 - val_loss: 1.1916 - val_accuracy: 0.6045
Epoch 37/200
80/80 [==============================] - 16s 205ms/step - loss: 0.8574 - accuracy: 0.
7064 - val_loss: 1.3812 - val_accuracy: 0.5700
Epoch 38/200
80/80 [==============================] - 16s 205ms/step - loss: 0.8734 - accuracy: 0.
6970 - val_loss: 1.1565 - val_accuracy: 0.6260
Epoch 39/200
80/80 [==============================] - 17s 208ms/step - loss: 0.8239 - accuracy: 0.
7147 - val_loss: 1.2140 - val_accuracy: 0.6100
Epoch 40/200
80/80 [==============================] - 17s 206ms/step - loss: 0.8300 - accuracy: 0.
7176 - val_loss: 1.2222 - val_accuracy: 0.6170
```

```
Epoch 41/200
80/80 [==============================] - 17s 206ms/step - loss: 0.8245 - accuracy: 0.
7181 - val_loss: 1.1498 - val_accuracy: 0.6355
Epoch 42/200
80/80 [==============================] - 17s 209ms/step - loss: 0.7953 - accuracy: 0.
7308 - val_loss: 1.2000 - val_accuracy: 0.6175
Epoch 43/200
80/80 [==============================] - 17s 208ms/step - loss: 0.7951 - accuracy: 0.
7300 - val_loss: 1.2458 - val_accuracy: 0.6130
Epoch 44/200
80/80 [==============================] - 17s 208ms/step - loss: 0.7860 - accuracy: 0.
7374 - val_loss: 1.3487 - val_accuracy: 0.5920
Epoch 45/200
80/80 [==============================] - 18s 230ms/step - loss: 0.7806 - accuracy: 0.
7343 - val_loss: 1.2513 - val_accuracy: 0.6025
Epoch 46/200
80/80 [==============================] - 18s 225ms/step - loss: 0.7832 - accuracy: 0.
7319 - val_loss: 1.2557 - val_accuracy: 0.6165
Epoch 47/200
80/80 [==============================] - 21s 255ms/step - loss: 0.7695 - accuracy: 0.
7351 - val_loss: 1.2520 - val_accuracy: 0.6160
Epoch 48/200
80/80 [==============================] - 17s 208ms/step - loss: 0.7220 - accuracy: 0.
7521 - val_loss: 1.3189 - val_accuracy: 0.6100
Epoch 49/200
80/80 [==============================] - 18s 228ms/step - loss: 0.7428 - accuracy: 0.
7440 - val_loss: 1.2230 - val_accuracy: 0.6135
Epoch 50/200
80/80 [==============================] - 21s 260ms/step - loss: 0.7164 - accuracy: 0.
7579 - val_loss: 1.2256 - val_accuracy: 0.6230
Epoch 51/200
80/80 [==============================] - 20s 253ms/step - loss: 0.6960 - accuracy: 0.
7628 - val_loss: 1.2627 - val_accuracy: 0.6280
Epoch 52/200
80/80 [==============================] - 19s 235ms/step - loss: 0.7007 - accuracy: 0.
7660 - val_loss: 1.2459 - val_accuracy: 0.6190
Epoch 53/200
80/80 [==============================] - 17s 206ms/step - loss: 0.6867 - accuracy: 0.
7678 - val_loss: 1.2470 - val_accuracy: 0.6405
Epoch 54/200
80/80 [==============================] - 20s 253ms/step - loss: 0.6737 - accuracy: 0.
7722 - val_loss: 1.3331 - val_accuracy: 0.6060
Epoch 55/200
80/80 [==============================] - 20s 251ms/step - loss: 0.6835 - accuracy: 0.
7671 - val_loss: 1.2238 - val_accuracy: 0.6330
Epoch 56/200
80/80 [==============================] - 20s 250ms/step - loss: 0.6552 - accuracy: 0.
7795 - val_loss: 1.2860 - val_accuracy: 0.6060
Epoch 57/200
80/80 [==============================] - 20s 254ms/step - loss: 0.6659 - accuracy: 0.
7768 - val_loss: 1.5034 - val_accuracy: 0.6000
Epoch 58/200
80/80 [==============================] - 20s 253ms/step - loss: 0.6674 - accuracy: 0.
7772 - val_loss: 1.2377 - val_accuracy: 0.6195
Epoch 59/200
80/80 [==============================] - 20s 251ms/step - loss: 0.6328 - accuracy: 0.
7854 - val_loss: 1.2401 - val_accuracy: 0.6435
Epoch 60/200
80/80 [==============================] - 21s 258ms/step - loss: 0.6480 - accuracy: 0.
7855 - val_loss: 1.2031 - val_accuracy: 0.6250
```

```
Epoch 61/200
80/80 [==============================] - 20s 251ms/step - loss: 0.6152 - accuracy: 0.
7950 - val_loss: 1.4266 - val_accuracy: 0.5995
Epoch 62/200
80/80 [==============================] - 20s 253ms/step - loss: 0.6071 - accuracy: 0.
7950 - val_loss: 1.2082 - val_accuracy: 0.6465
Epoch 63/200
80/80 [==============================] - 21s 256ms/step - loss: 0.6011 - accuracy: 0.
8012 - val_loss: 1.2845 - val_accuracy: 0.6215
Epoch 64/200
80/80 [==============================] - 21s 257ms/step - loss: 0.5885 - accuracy: 0.
8025 - val_loss: 1.2517 - val_accuracy: 0.6235
Epoch 65/200
80/80 [==============================] - 19s 236ms/step - loss: 0.5791 - accuracy: 0.
8014 - val_loss: 1.3445 - val_accuracy: 0.6225
Epoch 66/200
80/80 [==============================] - 19s 239ms/step - loss: 0.5761 - accuracy: 0.
8081 - val_loss: 1.2859 - val_accuracy: 0.6355
Epoch 67/200
80/80 [==============================] - 20s 255ms/step - loss: 0.5704 - accuracy: 0.
8090 - val_loss: 1.2663 - val_accuracy: 0.6350
Epoch 68/200
80/80 [==============================] - 16s 203ms/step - loss: 0.5773 - accuracy: 0.
8080 - val_loss: 1.3975 - val_accuracy: 0.6180
Epoch 69/200
80/80 [==============================] - 16s 203ms/step - loss: 0.5620 - accuracy: 0.
8105 - val_loss: 1.4119 - val_accuracy: 0.6235
Epoch 70/200
80/80 [==============================] - 17s 208ms/step - loss: 0.5617 - accuracy: 0.
8101 - val_loss: 1.2330 - val_accuracy: 0.6445
Epoch 71/200
80/80 [==============================] - 17s 208ms/step - loss: 0.5392 - accuracy: 0.
8201 - val_loss: 1.3711 - val_accuracy: 0.6240
Epoch 72/200
80/80 [==============================] - 17s 209ms/step - loss: 0.5271 - accuracy: 0.
8263 - val_loss: 1.3609 - val_accuracy: 0.6355
Epoch 73/200
80/80 [==============================] - 22s 270ms/step - loss: 0.5459 - accuracy: 0.
8198 - val_loss: 1.4466 - val_accuracy: 0.6090
Epoch 74/200
80/80 [==============================] - 17s 216ms/step - loss: 0.5337 - accuracy: 0.
8236 - val_loss: 1.3779 - val_accuracy: 0.6205
Epoch 75/200
80/80 [==============================] - 20s 244ms/step - loss: 0.5236 - accuracy: 0.
8317 - val_loss: 1.2974 - val_accuracy: 0.6405
Epoch 76/200
80/80 [==============================] - 22s 279ms/step - loss: 0.5000 - accuracy: 0.
8370 - val_loss: 1.4227 - val_accuracy: 0.6075
Epoch 77/200
80/80 [==============================] - 18s 221ms/step - loss: 0.5236 - accuracy: 0.
8289 - val_loss: 1.2994 - val_accuracy: 0.6310
Epoch 78/200
80/80 [==============================] - 17s 210ms/step - loss: 0.5113 - accuracy: 0.
8320 - val_loss: 1.3497 - val_accuracy: 0.6340
Epoch 79/200
80/80 [==============================] - 17s 208ms/step - loss: 0.4735 - accuracy: 0.
8444 - val_loss: 1.3659 - val_accuracy: 0.6475
Epoch 80/200
80/80 [==============================] - 17s 218ms/step - loss: 0.4763 - accuracy: 0.
8413 - val_loss: 1.4149 - val_accuracy: 0.6255
```

```
Epoch 81/200
80/80 [==============================] - 22s 272ms/step - loss: 0.4934 - accuracy: 0.
8382 - val_loss: 1.3104 - val_accuracy: 0.6385
Epoch 82/200
80/80 [==============================] - 22s 272ms/step - loss: 0.4646 - accuracy: 0.
8475 - val_loss: 1.3479 - val_accuracy: 0.6335
Epoch 83/200
80/80 [==============================] - 22s 274ms/step - loss: 0.4700 - accuracy: 0.
8468 - val_loss: 1.4432 - val_accuracy: 0.6230
Epoch 84/200
80/80 [==============================] - 22s 271ms/step - loss: 0.4647 - accuracy: 0.
8491 - val_loss: 1.4712 - val_accuracy: 0.6075
Epoch 85/200
80/80 [==============================] - 22s 271ms/step - loss: 0.4533 - accuracy: 0.
8561 - val_loss: 1.4292 - val_accuracy: 0.6225
Epoch 86/200
80/80 [==============================] - 19s 237ms/step - loss: 0.4546 - accuracy: 0.
8541 - val_loss: 1.3523 - val_accuracy: 0.6350
Epoch 87/200
80/80 [==============================] - 21s 263ms/step - loss: 0.4418 - accuracy: 0.
8540 - val_loss: 1.4631 - val_accuracy: 0.6205
Epoch 88/200
80/80 [==============================] - 22s 270ms/step - loss: 0.4591 - accuracy: 0.
8549 - val_loss: 1.4380 - val_accuracy: 0.6365
Epoch 89/200
80/80 [==============================] - 18s 225ms/step - loss: 0.4332 - accuracy: 0.
8633 - val_loss: 1.4225 - val_accuracy: 0.6285
Epoch 90/200
80/80 [==============================] - 21s 268ms/step - loss: 0.4439 - accuracy: 0.
8580 - val_loss: 1.3401 - val_accuracy: 0.6380
Epoch 91/200
80/80 [==============================] - 16s 205ms/step - loss: 0.4244 - accuracy: 0.
8620 - val_loss: 1.5009 - val_accuracy: 0.6225
Epoch 92/200
80/80 [==============================] - 20s 256ms/step - loss: 0.4142 - accuracy: 0.
8689 - val_loss: 1.4114 - val_accuracy: 0.6485
Epoch 93/200
80/80 [==============================] - 21s 269ms/step - loss: 0.4038 - accuracy: 0.
8719 - val_loss: 1.4307 - val_accuracy: 0.6415
Epoch 94/200
80/80 [==============================] - 17s 210ms/step - loss: 0.4205 - accuracy: 0.
8618 - val_loss: 1.4350 - val_accuracy: 0.6320
Epoch 95/200
80/80 [==============================] - 18s 220ms/step - loss: 0.4111 - accuracy: 0.
8689 - val_loss: 1.3754 - val_accuracy: 0.6350
Epoch 96/200
80/80 [==============================] - 22s 268ms/step - loss: 0.3923 - accuracy: 0.
8751 - val_loss: 1.6297 - val_accuracy: 0.6300
Epoch 97/200
80/80 [==============================] - 17s 206ms/step - loss: 0.3965 - accuracy: 0.
8758 - val_loss: 1.4123 - val_accuracy: 0.6420
Epoch 98/200
80/80 [==============================] - 20s 247ms/step - loss: 0.3936 - accuracy: 0.
8783 - val_loss: 1.5789 - val_accuracy: 0.6320
Epoch 99/200
80/80 [==============================] - 20s 249ms/step - loss: 0.3544 - accuracy: 0.
8884 - val_loss: 1.4931 - val_accuracy: 0.6320
Epoch 100/200
80/80 [==============================] - 19s 240ms/step - loss: 0.3926 - accuracy: 0.
8796 - val_loss: 1.4560 - val_accuracy: 0.6360
```

```
Epoch 101/200
80/80 [==============================] - 21s 260ms/step - loss: 0.3870 - accuracy: 0.
8783 - val_loss: 1.4717 - val_accuracy: 0.6420
Epoch 102/200
80/80 [==============================] - 28s 356ms/step - loss: 0.3888 - accuracy: 0.
8761 - val_loss: 1.4329 - val_accuracy: 0.6310
Epoch 103/200
80/80 [==============================] - 16s 196ms/step - loss: 0.3581 - accuracy: 0.
8855 - val_loss: 1.4172 - val_accuracy: 0.6400
Epoch 104/200
80/80 [==============================] - 16s 203ms/step - loss: 0.3619 - accuracy: 0.
8850 - val_loss: 1.5194 - val_accuracy: 0.6205
Epoch 105/200
80/80 [==============================] - 22s 280ms/step - loss: 0.3514 - accuracy: 0.
8891 - val_loss: 1.4574 - val_accuracy: 0.6490
Epoch 106/200
80/80 [==============================] - 21s 258ms/step - loss: 0.3619 - accuracy: 0.
8855 - val_loss: 1.4490 - val_accuracy: 0.6375
Epoch 107/200
80/80 [==============================] - 23s 279ms/step - loss: 0.3499 - accuracy: 0.
8929 - val_loss: 1.5347 - val_accuracy: 0.6305
Epoch 108/200
80/80 [==============================] - 22s 271ms/step - loss: 0.3210 - accuracy: 0.
8944 - val_loss: 1.4852 - val_accuracy: 0.6565
Epoch 109/200
80/80 [==============================] - 21s 257ms/step - loss: 0.3232 - accuracy: 0.
8990 - val_loss: 1.5321 - val_accuracy: 0.6435
Epoch 110/200
80/80 [==============================] - 21s 257ms/step - loss: 0.3410 - accuracy: 0.
8923 - val_loss: 1.4952 - val_accuracy: 0.6420
Epoch 111/200
80/80 [==============================] - 20s 253ms/step - loss: 0.3348 - accuracy: 0.
8917 - val_loss: 1.4300 - val_accuracy: 0.6445
Epoch 112/200
80/80 [==============================] - 21s 267ms/step - loss: 0.3361 - accuracy: 0.
8942 - val_loss: 1.4956 - val_accuracy: 0.6370
Epoch 113/200
80/80 [==============================] - 21s 261ms/step - loss: 0.3341 - accuracy: 0.
8970 - val_loss: 1.5228 - val_accuracy: 0.6375
Epoch 114/200
80/80 [==============================] - 21s 263ms/step - loss: 0.3168 - accuracy: 0.
9004 - val_loss: 1.5540 - val_accuracy: 0.6405
Epoch 115/200
80/80 [==============================] - 21s 266ms/step - loss: 0.3231 - accuracy: 0.
8994 - val_loss: 1.5093 - val_accuracy: 0.6400
Epoch 116/200
80/80 [==============================] - 21s 265ms/step - loss: 0.3452 - accuracy: 0.
8931 - val_loss: 1.5388 - val_accuracy: 0.6330
Epoch 117/200
80/80 [==============================] - 21s 262ms/step - loss: 0.3257 - accuracy: 0.
8989 - val_loss: 1.5094 - val_accuracy: 0.6455
Epoch 118/200
80/80 [==============================] - 22s 270ms/step - loss: 0.3182 - accuracy: 0.
8975 - val_loss: 1.5145 - val_accuracy: 0.6425
Epoch 119/200
80/80 [==============================] - 21s 261ms/step - loss: 0.2915 - accuracy: 0.
9064 - val_loss: 1.4981 - val_accuracy: 0.6450
Epoch 120/200
80/80 [==============================] - 20s 250ms/step - loss: 0.3101 - accuracy: 0.
9061 - val_loss: 1.6103 - val_accuracy: 0.6380
```

```
Epoch 121/200
80/80 [==============================] - 20s 250ms/step - loss: 0.3044 - accuracy: 0.
9075 - val_loss: 1.5981 - val_accuracy: 0.6375
Epoch 122/200
80/80 [==============================] - 21s 263ms/step - loss: 0.2997 - accuracy: 0.
9093 - val_loss: 1.5657 - val_accuracy: 0.6435
Epoch 123/200
80/80 [==============================] - 19s 244ms/step - loss: 0.3003 - accuracy: 0.
9055 - val_loss: 1.6278 - val_accuracy: 0.6370
Epoch 124/200
80/80 [==============================] - 20s 248ms/step - loss: 0.2936 - accuracy: 0.
9079 - val_loss: 1.6063 - val_accuracy: 0.6260
Epoch 125/200
80/80 [==============================] - 17s 214ms/step - loss: 0.2732 - accuracy: 0.
9147 - val_loss: 1.7406 - val_accuracy: 0.6270
Epoch 126/200
80/80 [==============================] - 15s 187ms/step - loss: 0.2963 - accuracy: 0.
9090 - val_loss: 1.6618 - val_accuracy: 0.6305
Epoch 127/200
80/80 [==============================] - 19s 232ms/step - loss: 0.2940 - accuracy: 0.
9100 - val_loss: 1.5980 - val_accuracy: 0.6385
Epoch 128/200
80/80 [==============================] - 17s 217ms/step - loss: 0.2727 - accuracy: 0.
9165 - val_loss: 1.7225 - val_accuracy: 0.6165
Epoch 129/200
80/80 [==============================] - 17s 216ms/step - loss: 0.2907 - accuracy: 0.
9090 - val_loss: 1.6485 - val_accuracy: 0.6275
Epoch 130/200
80/80 [==============================] - 17s 216ms/step - loss: 0.2762 - accuracy: 0.
9145 - val_loss: 1.6081 - val_accuracy: 0.6410
Epoch 131/200
80/80 [==============================] - 17s 215ms/step - loss: 0.2491 - accuracy: 0.
9224 - val_loss: 1.7411 - val_accuracy: 0.6365
Epoch 132/200
80/80 [==============================] - 18s 218ms/step - loss: 0.2711 - accuracy: 0.
9179 - val_loss: 1.7226 - val_accuracy: 0.6270
Epoch 133/200
80/80 [==============================] - 18s 219ms/step - loss: 0.2759 - accuracy: 0.
9154 - val_loss: 1.6320 - val_accuracy: 0.6320
Epoch 134/200
80/80 [==============================] - 17s 216ms/step - loss: 0.2749 - accuracy: 0.
9168 - val_loss: 1.6239 - val_accuracy: 0.6305
Epoch 135/200
80/80 [==============================] - 17s 215ms/step - loss: 0.2710 - accuracy: 0.
9159 - val_loss: 1.6855 - val_accuracy: 0.6340
Epoch 136/200
80/80 [==============================] - 17s 217ms/step - loss: 0.2529 - accuracy: 0.
9205 - val_loss: 1.7606 - val_accuracy: 0.6160
Epoch 137/200
80/80 [==============================] - 17s 215ms/step - loss: 0.2521 - accuracy: 0.
9219 - val_loss: 1.8214 - val_accuracy: 0.6155
Epoch 138/200
80/80 [==============================] - 18s 218ms/step - loss: 0.2738 - accuracy: 0.
9154 - val_loss: 1.5302 - val_accuracy: 0.6360
Epoch 139/200
80/80 [==============================] - 18s 227ms/step - loss: 0.2343 - accuracy: 0.
9291 - val_loss: 1.7946 - val_accuracy: 0.6390
Epoch 140/200
80/80 [==============================] - 18s 226ms/step - loss: 0.2514 - accuracy: 0.
9216 - val_loss: 1.5948 - val_accuracy: 0.6350
```

```
Epoch 141/200
80/80 [==============================] - 18s 219ms/step - loss: 0.2298 - accuracy: 0.
9274 - val_loss: 1.6827 - val_accuracy: 0.6410
Epoch 142/200
80/80 [==============================] - 17s 218ms/step - loss: 0.2570 - accuracy: 0.
9221 - val_loss: 1.7110 - val_accuracy: 0.6210
Epoch 143/200
80/80 [==============================] - 17s 217ms/step - loss: 0.2270 - accuracy: 0.
9315 - val_loss: 1.7621 - val_accuracy: 0.6235
Epoch 144/200
80/80 [==============================] - 18s 219ms/step - loss: 0.2237 - accuracy: 0.
9296 - val_loss: 1.6996 - val_accuracy: 0.6395
Epoch 145/200
80/80 [==============================] - 18s 227ms/step - loss: 0.2189 - accuracy: 0.
9337 - val_loss: 1.6170 - val_accuracy: 0.6550
Epoch 146/200
80/80 [==============================] - 18s 230ms/step - loss: 0.2274 - accuracy: 0.
9323 - val_loss: 1.9396 - val_accuracy: 0.6090
Epoch 147/200
80/80 [==============================] - 16s 195ms/step - loss: 0.2447 - accuracy: 0.
9266 - val_loss: 1.7496 - val_accuracy: 0.6370
Epoch 148/200
80/80 [==============================] - 17s 207ms/step - loss: 0.2570 - accuracy: 0.
9219 - val_loss: 1.7037 - val_accuracy: 0.6325
Epoch 149/200
80/80 [==============================] - 17s 215ms/step - loss: 0.2238 - accuracy: 0.
9314 - val_loss: 1.7555 - val_accuracy: 0.6300
Epoch 150/200
80/80 [==============================] - 15s 190ms/step - loss: 0.2287 - accuracy: 0.
9296 - val_loss: 1.6927 - val_accuracy: 0.6370
Epoch 151/200
80/80 [==============================] - 17s 207ms/step - loss: 0.2139 - accuracy: 0.
9346 - val_loss: 1.6790 - val_accuracy: 0.6420
Epoch 152/200
80/80 [==============================] - 17s 207ms/step - loss: 0.2288 - accuracy: 0.
9336 - val_loss: 1.7098 - val_accuracy: 0.6350
Epoch 153/200
80/80 [==============================] - 17s 209ms/step - loss: 0.2290 - accuracy: 0.
9311 - val_loss: 1.7551 - val_accuracy: 0.6365
Epoch 154/200
80/80 [==============================] - 16s 195ms/step - loss: 0.2295 - accuracy: 0.
9320 - val_loss: 1.6679 - val_accuracy: 0.6425
Epoch 155/200
80/80 [==============================] - 16s 201ms/step - loss: 0.2204 - accuracy: 0.
9330 - val_loss: 1.8664 - val_accuracy: 0.6185
Epoch 156/200
80/80 [==============================] - 16s 194ms/step - loss: 0.2325 - accuracy: 0.
9296 - val_loss: 1.9507 - val_accuracy: 0.6150
Epoch 157/200
80/80 [==============================] - 15s 188ms/step - loss: 0.2156 - accuracy: 0.
9354 - val_loss: 1.7460 - val_accuracy: 0.6410
Epoch 158/200
80/80 [==============================] - 16s 195ms/step - loss: 0.1917 - accuracy: 0.
9396 - val_loss: 1.7644 - val_accuracy: 0.6390
Epoch 159/200
80/80 [==============================] - 16s 201ms/step - loss: 0.2320 - accuracy: 0.
9310 - val_loss: 1.7394 - val_accuracy: 0.6385
Epoch 160/200
80/80 [==============================] - 16s 194ms/step - loss: 0.2085 - accuracy: 0.
9334 - val_loss: 1.7071 - val_accuracy: 0.6560
```

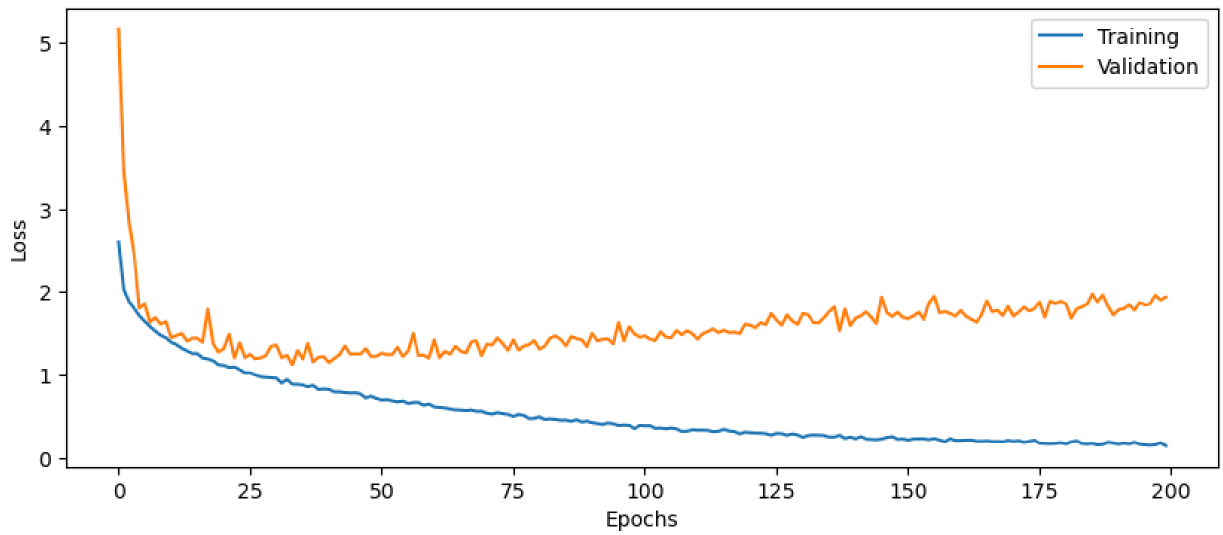
```
Epoch 161/200
80/80 [==============================] - 17s 207ms/step - loss: 0.2093 - accuracy: 0.
9358 - val_loss: 1.7762 - val_accuracy: 0.6250
Epoch 162/200
80/80 [==============================] - 16s 196ms/step - loss: 0.2133 - accuracy: 0.
9337 - val_loss: 1.7057 - val_accuracy: 0.6360
Epoch 163/200
80/80 [==============================] - 15s 192ms/step - loss: 0.2132 - accuracy: 0.
9376 - val_loss: 1.6689 - val_accuracy: 0.6330
Epoch 164/200
80/80 [==============================] - 15s 181ms/step - loss: 0.2007 - accuracy: 0.
9416 - val_loss: 1.6345 - val_accuracy: 0.6540
Epoch 165/200
80/80 [==============================] - 14s 180ms/step - loss: 0.1985 - accuracy: 0.
9404 - val_loss: 1.7241 - val_accuracy: 0.6245
Epoch 166/200
80/80 [==============================] - 15s 187ms/step - loss: 0.2030 - accuracy: 0.
9365 - val_loss: 1.8936 - val_accuracy: 0.6365
Epoch 167/200
80/80 [==============================] - 16s 193ms/step - loss: 0.1963 - accuracy: 0.
9416 - val_loss: 1.7610 - val_accuracy: 0.6440
Epoch 168/200
80/80 [==============================] - 16s 201ms/step - loss: 0.1970 - accuracy: 0.
9408 - val_loss: 1.7782 - val_accuracy: 0.6430
Epoch 169/200
80/80 [==============================] - 16s 193ms/step - loss: 0.1935 - accuracy: 0.
9421 - val_loss: 1.7132 - val_accuracy: 0.6480
Epoch 170/200
80/80 [==============================] - 16s 196ms/step - loss: 0.2078 - accuracy: 0.
9367 - val_loss: 1.8269 - val_accuracy: 0.6360
Epoch 171/200
80/80 [==============================] - 15s 189ms/step - loss: 0.1976 - accuracy: 0.
9431 - val_loss: 1.7080 - val_accuracy: 0.6445
Epoch 172/200
80/80 [==============================] - 15s 180ms/step - loss: 0.2052 - accuracy: 0.
9384 - val_loss: 1.7561 - val_accuracy: 0.6250
Epoch 173/200
80/80 [==============================] - 16s 197ms/step - loss: 0.1863 - accuracy: 0.
9434 - val_loss: 1.8176 - val_accuracy: 0.6515
Epoch 174/200
80/80 [==============================] - 15s 181ms/step - loss: 0.1976 - accuracy: 0.
9406 - val_loss: 1.7700 - val_accuracy: 0.6240
Epoch 175/200
80/80 [==============================] - 16s 204ms/step - loss: 0.2105 - accuracy: 0.
9370 - val_loss: 1.8004 - val_accuracy: 0.6425
Epoch 176/200
80/80 [==============================] - 17s 215ms/step - loss: 0.1773 - accuracy: 0.
9469 - val_loss: 1.8784 - val_accuracy: 0.6405
Epoch 177/200
80/80 [==============================] - 15s 186ms/step - loss: 0.1728 - accuracy: 0.
9495 - val_loss: 1.6948 - val_accuracy: 0.6405
Epoch 178/200
80/80 [==============================] - 15s 181ms/step - loss: 0.1710 - accuracy: 0.
9481 - val_loss: 1.8901 - val_accuracy: 0.6305
Epoch 179/200
80/80 [==============================] - 15s 188ms/step - loss: 0.1741 - accuracy: 0.
9495 - val_loss: 1.8595 - val_accuracy: 0.6340
Epoch 180/200
80/80 [==============================] - 14s 178ms/step - loss: 0.1819 - accuracy: 0.
9460 - val_loss: 1.8884 - val_accuracy: 0.6420
```

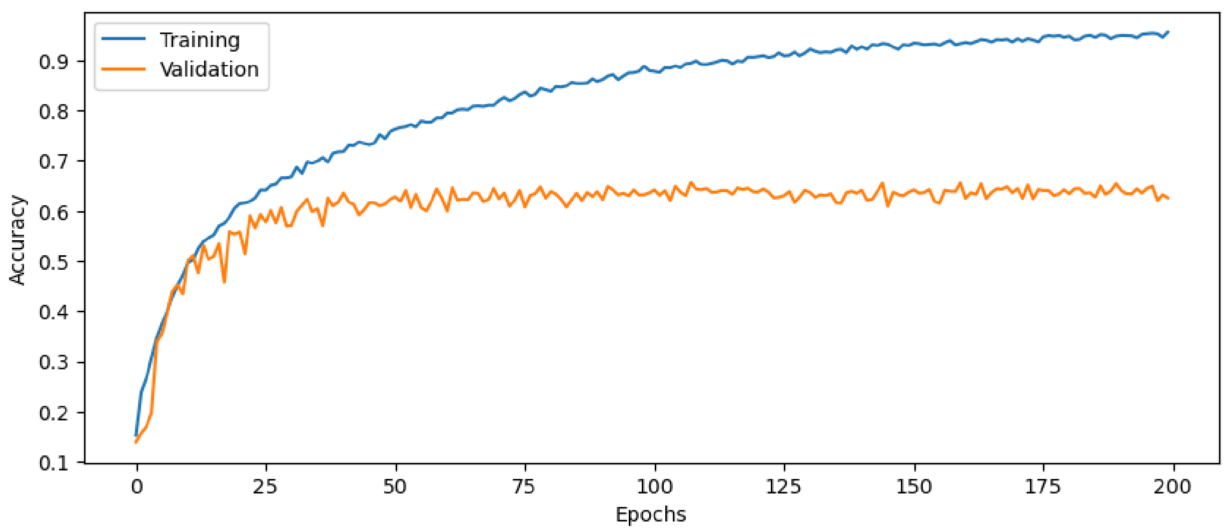
```
Epoch 181/200
80/80 [==============================] - 16s 194ms/step - loss: 0.1683 - accuracy: 0.
9482 - val_loss: 1.8582 - val_accuracy: 0.6340
Epoch 182/200
80/80 [==============================] - 15s 191ms/step - loss: 0.1928 - accuracy: 0.
9409 - val_loss: 1.6832 - val_accuracy: 0.6440
Epoch 183/200
80/80 [==============================] - 15s 193ms/step - loss: 0.2037 - accuracy: 0.
9421 - val_loss: 1.7955 - val_accuracy: 0.6445
Epoch 184/200
80/80 [==============================] - 15s 185ms/step - loss: 0.1729 - accuracy: 0.
9482 - val_loss: 1.8207 - val_accuracy: 0.6350
Epoch 185/200
80/80 [==============================] - 14s 180ms/step - loss: 0.1675 - accuracy: 0.
9500 - val_loss: 1.8541 - val_accuracy: 0.6360
Epoch 186/200
80/80 [==============================] - 15s 182ms/step - loss: 0.1748 - accuracy: 0.
9459 - val_loss: 1.9799 - val_accuracy: 0.6275
Epoch 187/200
80/80 [==============================] - 16s 195ms/step - loss: 0.1582 - accuracy: 0.
9515 - val_loss: 1.8827 - val_accuracy: 0.6500
Epoch 188/200
80/80 [==============================] - 16s 194ms/step - loss: 0.1630 - accuracy: 0.
9500 - val_loss: 1.9643 - val_accuracy: 0.6330
Epoch 189/200
80/80 [==============================] - 15s 186ms/step - loss: 0.1889 - accuracy: 0.
9434 - val_loss: 1.8288 - val_accuracy: 0.6395
Epoch 190/200
80/80 [==============================] - 15s 181ms/step - loss: 0.1784 - accuracy: 0.
9481 - val_loss: 1.7209 - val_accuracy: 0.6540
Epoch 191/200
80/80 [==============================] - 15s 183ms/step - loss: 0.1659 - accuracy: 0.
9498 - val_loss: 1.7861 - val_accuracy: 0.6405
Epoch 192/200
80/80 [==============================] - 15s 189ms/step - loss: 0.1779 - accuracy: 0.
9494 - val_loss: 1.7992 - val_accuracy: 0.6345
Epoch 193/200
80/80 [==============================] - 18s 219ms/step - loss: 0.1684 - accuracy: 0.
9490 - val_loss: 1.8476 - val_accuracy: 0.6335
Epoch 194/200
80/80 [==============================] - 15s 184ms/step - loss: 0.1874 - accuracy: 0.
9451 - val_loss: 1.7809 - val_accuracy: 0.6440
Epoch 195/200
80/80 [==============================] - 15s 191ms/step - loss: 0.1641 - accuracy: 0.
9519 - val_loss: 1.8732 - val_accuracy: 0.6350
Epoch 196/200
80/80 [==============================] - 15s 191ms/step - loss: 0.1581 - accuracy: 0.
9529 - val_loss: 1.8435 - val_accuracy: 0.6445
Epoch 197/200
80/80 [==============================] - 15s 188ms/step - loss: 0.1518 - accuracy: 0.
9540 - val_loss: 1.8576 - val_accuracy: 0.6490
Epoch 198/200
80/80 [==============================] - 16s 194ms/step - loss: 0.1588 - accuracy: 0.
9528 - val_loss: 1.9602 - val_accuracy: 0.6205
Epoch 199/200
80/80 [==============================] - 14s 178ms/step - loss: 0.1823 - accuracy: 0.
9457 - val_loss: 1.9044 - val_accuracy: 0.6325
Epoch 200/200
80/80 [==============================] - 14s 181ms/step - loss: 0.1450 - accuracy: 0.
9561 - val_loss: 1.9395 - val_accuracy: 0.6260
```

In [22]:
```
# 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: 1.9383
Test accuracy: 0.6410
Test loss: 4.7774
Test accuracy: 0.2820
```

In [23]:
```
# 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 [24]:
```
# 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 [==============================] - 1s 12ms/step
```

In [25]:
```
# 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?

- No, a CNN trained on 32 x 32 images can't be directly applied to images of another size. CNN architecture, including filter sizes and pooling layers, is tailored to specific input dimensions. Adapting to different sizes would require architectural modifications and may lead to ineffective feature representations without retraining.

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?

- Yes, it is possible to design a CNN that can be trained on images of one size and then applied to images of any size using techniques such as, Global Average Pooling,Fully Convolutional Networks

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?

- 5 layers

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

- Trainable params: 25,583,592

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

- 224 x 224

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

- It will increase the number of Trainable params multiplicatively, and it will be extra horrofying when we have too much nodes in layers

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 [10]:

```
# 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 keras.applications import ResNet50

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

# Display the model summary
modelres.summary()
```

```
Model: "resnet50"
__________________________________________________________________________________
________________
 Layer (type)                   Output Shape         Param #     Connected to
==================================================================================
================
 input_2 (InputLayer)           [(None, 224, 224, 3  0           []
                                )]

 conv1_pad (ZeroPadding2D)      (None, 230, 230, 3)  0           ['input_2[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 [20]:
```python
import numpy as np
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.resnet50 import preprocess_input, decode_predictior

def model_predict(img_path):

    img=image.load_img(img_path,target_size=(224, 224))
    x = image.img_to_array(img)
    x = np.expand_dims(x, axis=0)
    x = preprocess_input(x)

    preds = modelres.predict(x)
    decoded_predictions = decode_predictions(preds, top=3)[0]
    return decoded_predictions
```

- I have added sport car,kingfisher, deer,golden retriever and horse respectively.

In [21]:
```python
model_predict(  'C:/Users/PC/Documents/Python_Scripts/deep_learningl2/img1.jpg')
```

```
1/1 [==============================] - 9s 9s/step
Downloading data from https://storage.googleapis.com/download.tensorflow.org/data/ima
genet_class_index.json
35363/35363 [==============================] - 0s 0us/step
```

Out[21]:
```
[('n02814533', 'beach_wagon', 0.33461186),
 ('n02974003', 'car_wheel', 0.32645342),
 ('n04285008', 'sports_car', 0.13993666)]
```

In [22]:
```python
model_predict(  'C:/Users/PC/Documents/Python_Scripts/deep_learningl2/img2.jpg')
```

```
1/1 [==============================] - 0s 222ms/step
```

Out[22]:
```
[('n01828970', 'bee_eater', 0.9071793),
 ('n01843065', 'jacamar', 0.022781594),
 ('n04404412', 'television', 0.017308157)]
```

In [23]:
```python
model_predict(  'C:/Users/PC/Documents/Python_Scripts/deep_learningl2/img3.jpg')
```

```
1/1 [==============================] - 0s 175ms/step
```

Out[23]:
```
[('n02417914', 'ibex', 0.88857865),
 ('n02422106', 'hartebeest', 0.051604267),
 ('n02115913', 'dhole', 0.034042176)]
```

In [24]:
```python
model_predict(  'C:/Users/PC/Documents/Python_Scripts/deep_learningl2/img4.jpg')
```

```
1/1 [==============================] - 0s 194ms/step
```

Out[24]:
```
[('n02099601', 'golden_retriever', 0.89418375),
 ('n02099712', 'Labrador_retriever', 0.08991353),
 ('n02101556', 'clumber', 0.006692364)]
```

In [25]:
```python
model_predict(  'C:/Users/PC/Documents/Python_Scripts/deep_learningl2/img5.jpg')
```

```
1/1 [==============================] - 0s 170ms/step
```

Out[25]:

```
[('n02389026', 'sorrel', 0.9190898),
 ('n02422106', 'hartebeest', 0.0474683),
 ('n06359193', 'web_site', 0.008529172)]
```