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).

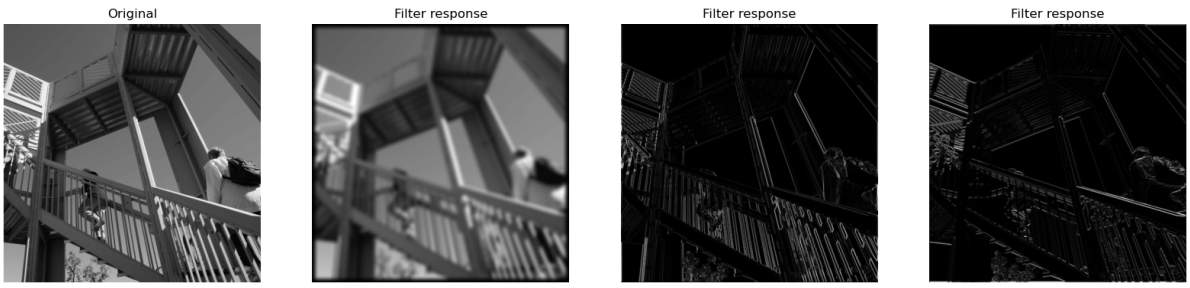
In [61]:
```
# This cell is finished

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\ericchenta\AppData\Local\Temp\ipykernel_117472\2994295117.py:8: Dep
recationWarning: scipy.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 scipy.datasets.ascent instead.
  image = misc.ascent()
```

In [2]:
```
# Perform convolution using the function 'convolve2d' for the different filt
filterResponseGauss = signal.convolve2d(image,gaussFilter)
filterResponseSobelX = signal.convolve2d(image,sobelX)
filterResponseSobelY = signal.convolve2d(image,sobelY)
```

In [3]:
```
import matplotlib.pyplot as plt

# Show filter responses
fig, (ax_orig, ax_filt1, ax_filt2, ax_filt3) = plt.subplots(1, 4, figsize=(2
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?

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?

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

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

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?

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

In [62]:
```
# Your code for checking sizes of image and filter responses
#Q2
print(image.shape)
#Q3
print(gaussFilter.shape)
print(sobelX.shape)
print(sobelY.shape)
#Q4
print(signal.convolve2d(image, sobelX,mode='same').shape)
#Q5
print(signal.convolve2d(image, gaussFilter,mode='valid').shape)
print(signal.convolve2d(image, sobelX,mode='valid').shape)
print(signal.convolve2d(image, sobelY,mode='valid').shape)
```

```
(512, 512)
(15, 15)
(3, 3)
(3, 3)
(512, 512)
(498, 498)
(510, 510)
(510, 510)
```

Answers

- A1 : SobelX and SobelY are filters that extract vertical lines and horizontal lines in the image,respectively. The gaussian filter blurs the image.
- A2 : Size of original image is (512,512). Given that the shape of the image is 2 Dimensional, it is single channeled. A color image has 3 channels, one for each of RGB.
- A3 :
  - Gaussian has shape (15,15)
  - SobelX has shape (3,3)
  - SobelY has shape (3,3)
- A4 : if same is used the size of the filter response is (512,512) - same as the input image.
- A5 : If mode 'valid' is used Gaussian filter response will have the size 498x498. SobelX and SobelY responses will have size 510x510. If the size of valid filter increase, the response size will decrease.
- A6 : Valid convolutions are a problems because in a CNN with many layers, there many be many values that are zero, which will cause the filter response to decrease in size with each layer. This could cause the output image to be much smaller in dimension compared to the input image.

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 [5]:
```
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 mem
#physical_devices = tf.config.experimental.list_physical_devices('GPU')
#tf.config.experimental.set_memory_growth(physical_devices[0], True)
```

Part 4: How fast is the graphics card?

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

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?

Question 9: Do you think that a graphics card, compared to the CPU, is equally faster for convolving a batch of 1,000 images, compared to convolving a batch of 3 images? Motivate your answer.

Answer

- A7 : This is because a colour image has 3 channels, RedGreenBlue(RGB).
- A8 : Although Conv2D and signal.convolve2d perform similar task(apply kernels or filters to the image). However, in Conv2D the kernels are trainable and can be optimized during training.
- A9 : We will say that it will not be equally faster. As GPU is optimized for parallel computations and have way more cores than GPU. Hence if the batch size is larger, the advantage over CPU will be more significant.

Part 5: Load data

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

In [6]:
```
from keras.datasets import cifar10
import numpy as np
```

```
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 's

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

print("Training images have size {} and labels have size {} ".format(Xtrain.
print("Test images have size {} and labels have size {} \n ".format(Xtest.sh

# Reduce the number of images for training and testing to 10000 and 2000 res
# 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 " % (Xtr
print("Reduced test images have size %s and labels have size %s \n" % (Xtest

# 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(
```

```
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 (1
0000, 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 [7]:
```
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 [8]:
```
from sklearn.model_selection import train_test_split

# Your code for splitting the dataset
Xtrain,Xval,Ytrain,Yval = train_test_split(Xtrain,Ytrain,test_size= 0.25)

# Print the size of training data, validation data and test data
print(Xtrain.shape)
print(Ytrain.shape)
print(Xval.shape)
print(Yval.shape)
print(Xtest.shape)
print(Ytest.shape)
```

```
(7500, 32, 32, 3)
(7500, 1)
(2500, 32, 32, 3)
(2500, 1)
(2000, 32, 32, 3)
(2000, 1)
```

Part 8: Preprocessing of images

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

In [9]:
```
# 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 [10]:
```
from tensorflow.keras.utils import to_categorical

# Print shapes before converting the labels
print(Ytrain.shape)
print(Yval.shape)
print(Ytest.shape)

# Your code for converting Ytrain, Yval, Ytest to categorical
Ytrain = to_categorical(Ytrain, num_classes=10, dtype="float32")
Yval = to_categorical(Yval, num_classes=10, dtype="float32")
Ytest = to_categorical(Ytest, num_classes=10, dtype="float32")

# Print shapes after converting the labels
print(Ytrain.shape)
print(Yval.shape)
print(Ytest.shape)
```

```
(7500, 1)
(2500, 1)
(2000, 1)
(7500, 10)
(2500, 10)
(2000, 10)
```

Part 10: 2D CNN

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

Relevant functions are

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

`Dense()`, a dense network layer

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

`BatchNormalization()`, perform batch normalization

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

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

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

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

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

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

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

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

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

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

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

In [11]:
```
import tensorflow as tf
from keras.models import Sequential, Model
from keras.layers import Input, Conv2D, BatchNormalization, MaxPooling2D, Fl
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,

    # Setup a sequential model
    model = Sequential()

    # Add first convolutional layer to the model, requires input shape
    model.add(Input(shape= input_shape))

    # Add remaining convolutional layers to the model, the number of filters
    for i in range(n_conv_layers): # -1 in range ALTERNATIVE
        model.add(tf.keras.layers.Conv2D(n_filters*2**(i),kernel_size=(3,3),
        model.add(BatchNormalization())
        model.add(tf.keras.layers.MaxPooling2D(pool_size=(2, 2))) # is paddi

    # Add flatten layer
    model.add(tf.keras.layers.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):
            model.add(Dropout(0.5))
    # Add final dense layer
```

```
    model.add(Dense(10,activation = 'softmax'))


    # Compile model
    model.compile(optimizer='adam',loss= 'CategoricalCrossentropy',metrics =

    return model
```

In [12]:
```
# 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 [13]:
```
# 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_filters=16,n_dense_layers=0

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

```
Epoch 1/20
75/75 [==============================] - 2s 25ms/step - loss: 2.0262 - accur
acy: 0.3393 - val_loss: 2.0444 - val_accuracy: 0.3160
Epoch 2/20
75/75 [==============================] - 1s 19ms/step - loss: 1.4814 - accur
acy: 0.4860 - val_loss: 1.8939 - val_accuracy: 0.3288
Epoch 3/20
75/75 [==============================] - 2s 23ms/step - loss: 1.2892 - accur
acy: 0.5484 - val_loss: 1.7316 - val_accuracy: 0.3724
Epoch 4/20
75/75 [==============================] - 2s 21ms/step - loss: 1.1375 - accur
acy: 0.6011 - val_loss: 1.6454 - val_accuracy: 0.4104
Epoch 5/20
75/75 [==============================] - 1s 20ms/step - loss: 1.0287 - accur
acy: 0.6337 - val_loss: 1.5694 - val_accuracy: 0.4352
Epoch 6/20
75/75 [==============================] - 1s 18ms/step - loss: 0.9332 - accur
acy: 0.6753 - val_loss: 1.4153 - val_accuracy: 0.4984
Epoch 7/20
75/75 [==============================] - 1s 18ms/step - loss: 0.8492 - accur
acy: 0.7029 - val_loss: 1.4231 - val_accuracy: 0.5220
Epoch 8/20
75/75 [==============================] - 2s 23ms/step - loss: 0.7629 - accur
acy: 0.7312 - val_loss: 1.4089 - val_accuracy: 0.5336
Epoch 9/20
75/75 [==============================] - 2s 20ms/step - loss: 0.6962 - accur
acy: 0.7609 - val_loss: 1.4178 - val_accuracy: 0.5368
Epoch 10/20
75/75 [==============================] - 1s 20ms/step - loss: 0.6269 - accur
acy: 0.7885 - val_loss: 1.4576 - val_accuracy: 0.5408
Epoch 11/20
75/75 [==============================] - 2s 23ms/step - loss: 0.5733 - accur
acy: 0.8052 - val_loss: 1.5458 - val_accuracy: 0.5348
Epoch 12/20
75/75 [==============================] - 2s 23ms/step - loss: 0.5174 - accur
acy: 0.8291 - val_loss: 1.4918 - val_accuracy: 0.5432
Epoch 13/20
75/75 [==============================] - 1s 19ms/step - loss: 0.4550 - accur
acy: 0.8525 - val_loss: 1.5696 - val_accuracy: 0.5448
Epoch 14/20
75/75 [==============================] - 2s 22ms/step - loss: 0.4153 - accur
acy: 0.8687 - val_loss: 1.6337 - val_accuracy: 0.5428
Epoch 15/20
75/75 [==============================] - 2s 20ms/step - loss: 0.3717 - accur
acy: 0.8863 - val_loss: 1.7269 - val_accuracy: 0.5320
Epoch 16/20
75/75 [==============================] - 2s 21ms/step - loss: 0.3302 - accur
acy: 0.9000 - val_loss: 1.7524 - val_accuracy: 0.5312
Epoch 17/20
75/75 [==============================] - 1s 20ms/step - loss: 0.2965 - accur
acy: 0.9103 - val_loss: 1.8029 - val_accuracy: 0.5280
Epoch 18/20
75/75 [==============================] - 1s 19ms/step - loss: 0.2593 - accur
acy: 0.9264 - val_loss: 1.8768 - val_accuracy: 0.5388
Epoch 19/20
75/75 [==============================] - 2s 32ms/step - loss: 0.2315 - accur
acy: 0.9344 - val_loss: 1.8920 - val_accuracy: 0.5488
Epoch 20/20
75/75 [==============================] - 3s 45ms/step - loss: 0.1964 - accur
acy: 0.9512 - val_loss: 1.9563 - val_accuracy: 0.5332
```

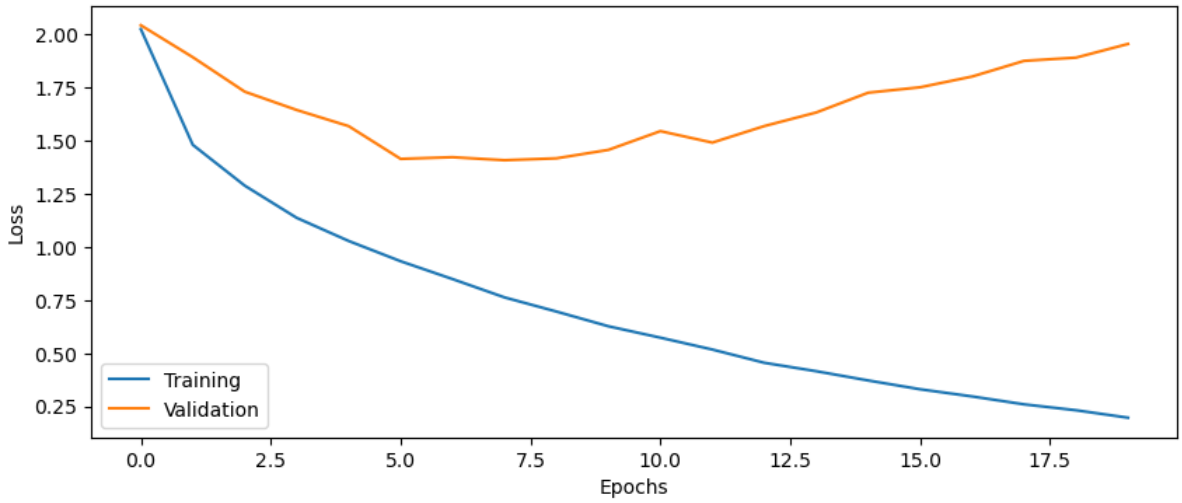
In [14]:

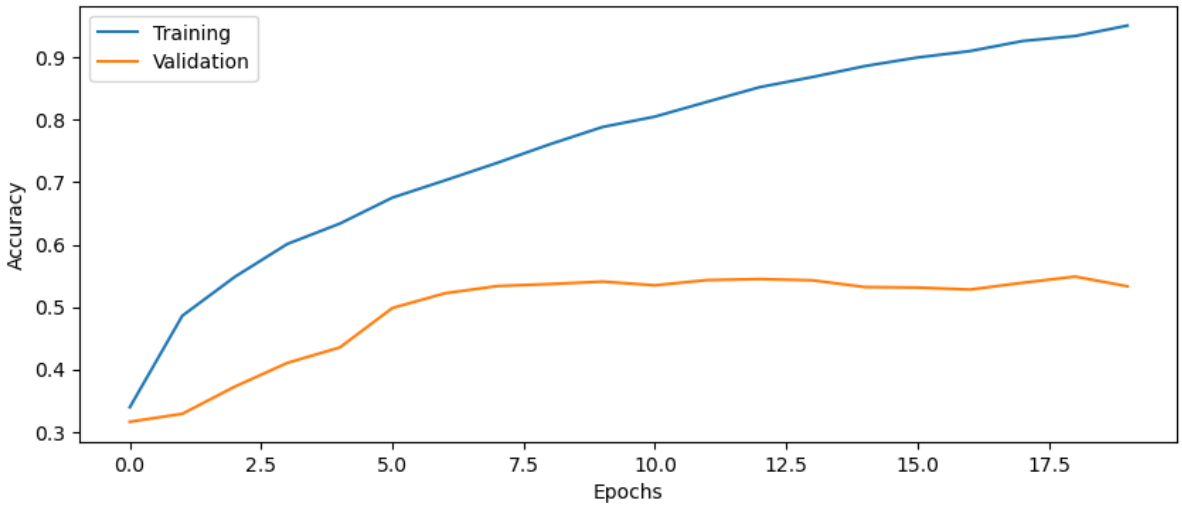
```
# Evaluate the trained model on test set, not used in training or validation
score = model1.evaluate(x = Xtest,y = Ytest)
```

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

```
63/63 [==============================] - 1s 5ms/step - loss: 1.7804 - accura
cy: 0.5650
Test loss: 1.7804
Test accuracy: 0.5650
```

In [15]:
```
# Plot the history from the training run
plot_results(history1)
```





Part 12: Improving performance

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

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

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?

Answers

- A10 : Training accuracy is 96.12% and test accuracy is 55.55% leading to a difference of ~40%.

- A11 : Because the input data has larger dimension and also in Conv2D other parameters such as filters are also trainable, so we have to decrease batch size to fit in the Memory.

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

In [16]:
```
# 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_filters=16,n_dense_layers=1

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

```
Epoch 1/20
75/75 [==============================] - 3s 35ms/step - loss: 1.6945 - accur
acy: 0.3993 - val_loss: 2.2779 - val_accuracy: 0.1684
Epoch 2/20
75/75 [==============================] - 2s 28ms/step - loss: 1.2605 - accur
acy: 0.5543 - val_loss: 2.5372 - val_accuracy: 0.1576
Epoch 3/20
75/75 [==============================] - 2s 23ms/step - loss: 1.0494 - accur
acy: 0.6395 - val_loss: 2.5164 - val_accuracy: 0.1880
Epoch 4/20
75/75 [==============================] - 2s 23ms/step - loss: 0.8672 - accur
acy: 0.7073 - val_loss: 2.3838 - val_accuracy: 0.2344
Epoch 5/20
75/75 [==============================] - 1s 18ms/step - loss: 0.7075 - accur
acy: 0.7760 - val_loss: 2.0687 - val_accuracy: 0.3184
Epoch 6/20
75/75 [==============================] - 2s 33ms/step - loss: 0.5568 - accur
acy: 0.8344 - val_loss: 1.7049 - val_accuracy: 0.4324
Epoch 7/20
75/75 [==============================] - 2s 23ms/step - loss: 0.4288 - accur
acy: 0.8813 - val_loss: 1.5000 - val_accuracy: 0.4992
Epoch 8/20
75/75 [==============================] - 2s 23ms/step - loss: 0.3196 - accur
acy: 0.9245 - val_loss: 1.4881 - val_accuracy: 0.5156
Epoch 9/20
75/75 [==============================] - 2s 21ms/step - loss: 0.2371 - accur
acy: 0.9496 - val_loss: 1.5456 - val_accuracy: 0.5184
Epoch 10/20
75/75 [==============================] - 2s 26ms/step - loss: 0.1601 - accur
acy: 0.9801 - val_loss: 1.5753 - val_accuracy: 0.5312
Epoch 11/20
75/75 [==============================] - 2s 23ms/step - loss: 0.1078 - accur
acy: 0.9901 - val_loss: 1.6411 - val_accuracy: 0.5308
Epoch 12/20
75/75 [==============================] - 1s 20ms/step - loss: 0.0829 - accur
acy: 0.9932 - val_loss: 1.7528 - val_accuracy: 0.5312
Epoch 13/20
75/75 [==============================] - 2s 22ms/step - loss: 0.0591 - accur
acy: 0.9965 - val_loss: 1.7898 - val_accuracy: 0.5380
Epoch 14/20
75/75 [==============================] - 2s 25ms/step - loss: 0.0405 - accur
acy: 0.9987 - val_loss: 1.8069 - val_accuracy: 0.5324
Epoch 15/20
75/75 [==============================] - 2s 28ms/step - loss: 0.0285 - accur
acy: 0.9993 - val_loss: 1.8640 - val_accuracy: 0.5256
Epoch 16/20
75/75 [==============================] - 2s 21ms/step - loss: 0.0228 - accur
acy: 0.9996 - val_loss: 1.9138 - val_accuracy: 0.5300
Epoch 17/20
75/75 [==============================] - 1s 19ms/step - loss: 0.0186 - accur
acy: 0.9999 - val_loss: 1.9420 - val_accuracy: 0.5296
Epoch 18/20
75/75 [==============================] - 2s 21ms/step - loss: 0.0149 - accur
acy: 1.0000 - val_loss: 1.9674 - val_accuracy: 0.5332
Epoch 19/20
75/75 [==============================] - 1s 17ms/step - loss: 0.0116 - accur
acy: 1.0000 - val_loss: 2.0355 - val_accuracy: 0.5248
Epoch 20/20
75/75 [==============================] - 1s 16ms/step - loss: 0.0095 - accur
acy: 1.0000 - val_loss: 2.0465 - val_accuracy: 0.5268
```

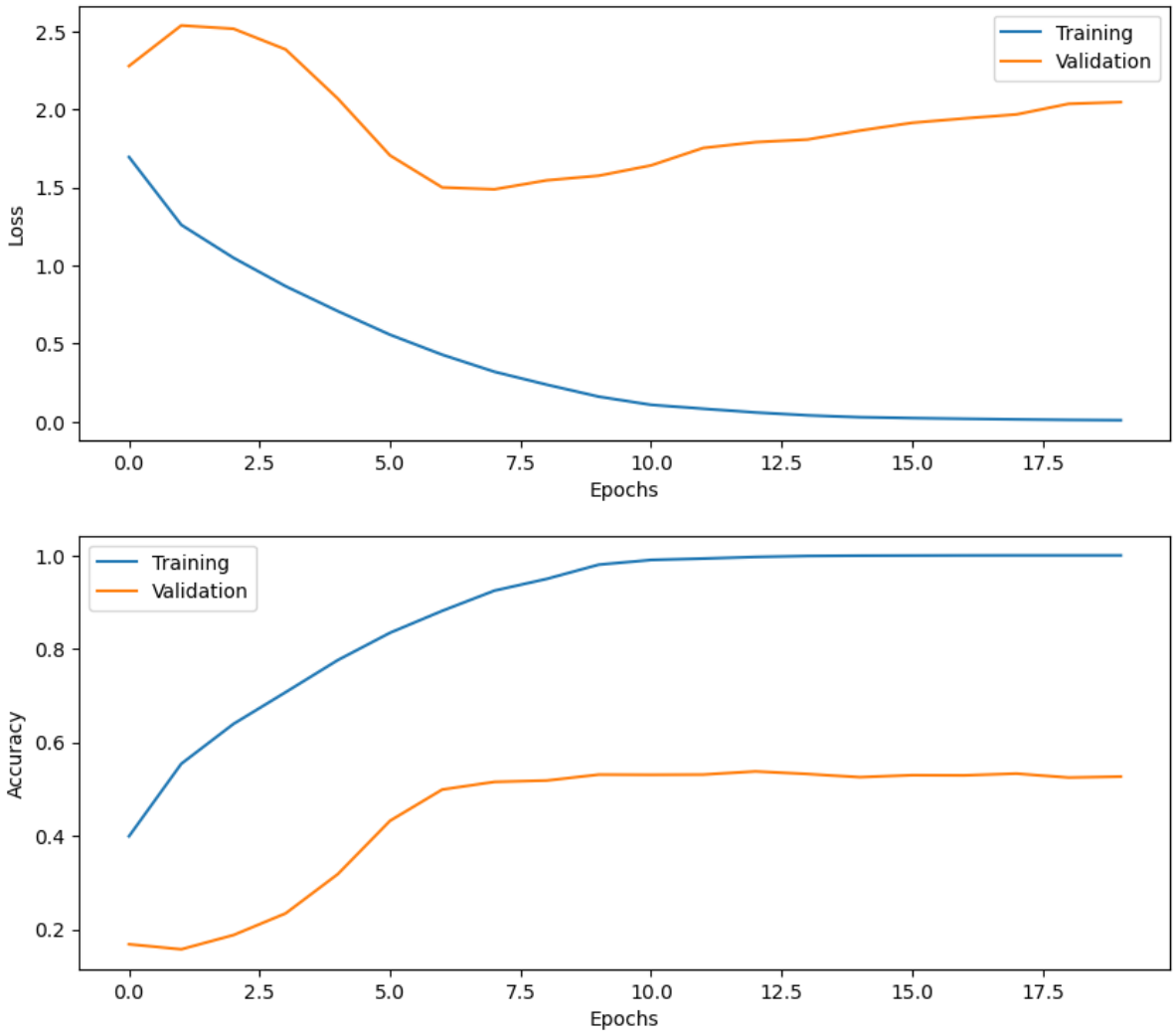
In [17]:

```
# Evaluate the trained model on test set, not used in training or validation
score = model2.evaluate(Xtest,Ytest)
```

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

```
63/63 [==============================] - 1s 7ms/step - loss: 1.9728 - accura
cy: 0.5300
Test loss: 1.9728
Test accuracy: 0.5300
```

In [18]:
```
# Plot the history from the training run
plot_results(history2)
```



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

In [19]:
```
# 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_filters=16,n_dense_layers=1

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

```
Epoch 1/20
75/75 [==============================] - 2s 24ms/step - loss: 1.6981 - accur
acy: 0.4008 - val_loss: 2.4411 - val_accuracy: 0.1036
Epoch 2/20
75/75 [==============================] - 1s 16ms/step - loss: 1.2135 - accur
acy: 0.5761 - val_loss: 2.6042 - val_accuracy: 0.1640
Epoch 3/20
75/75 [==============================] - 1s 18ms/step - loss: 0.9626 - accur
acy: 0.6693 - val_loss: 2.7644 - val_accuracy: 0.1768
Epoch 4/20
75/75 [==============================] - 1s 19ms/step - loss: 0.7473 - accur
acy: 0.7499 - val_loss: 2.7005 - val_accuracy: 0.2244
Epoch 5/20
75/75 [==============================] - 2s 23ms/step - loss: 0.5333 - accur
acy: 0.8367 - val_loss: 2.0923 - val_accuracy: 0.3456
Epoch 6/20
75/75 [==============================] - 1s 20ms/step - loss: 0.3498 - accur
acy: 0.9064 - val_loss: 1.8013 - val_accuracy: 0.4340
Epoch 7/20
75/75 [==============================] - 1s 19ms/step - loss: 0.2232 - accur
acy: 0.9491 - val_loss: 1.6190 - val_accuracy: 0.4896
Epoch 8/20
75/75 [==============================] - 1s 19ms/step - loss: 0.1423 - accur
acy: 0.9753 - val_loss: 1.6727 - val_accuracy: 0.5116
Epoch 9/20
75/75 [==============================] - 1s 18ms/step - loss: 0.0726 - accur
acy: 0.9928 - val_loss: 1.6317 - val_accuracy: 0.5432
Epoch 10/20
75/75 [==============================] - 2s 21ms/step - loss: 0.0434 - accur
acy: 0.9977 - val_loss: 1.6916 - val_accuracy: 0.5400
Epoch 11/20
75/75 [==============================] - 1s 19ms/step - loss: 0.0223 - accur
acy: 0.9997 - val_loss: 1.7282 - val_accuracy: 0.5412
Epoch 12/20
75/75 [==============================] - 1s 17ms/step - loss: 0.0121 - accur
acy: 1.0000 - val_loss: 1.7225 - val_accuracy: 0.5476
Epoch 13/20
75/75 [==============================] - 2s 20ms/step - loss: 0.0086 - accur
acy: 1.0000 - val_loss: 1.7468 - val_accuracy: 0.5560
Epoch 14/20
75/75 [==============================] - 2s 20ms/step - loss: 0.0064 - accur
acy: 1.0000 - val_loss: 1.7608 - val_accuracy: 0.5564
Epoch 15/20
75/75 [==============================] - 2s 20ms/step - loss: 0.0054 - accur
acy: 1.0000 - val_loss: 1.7800 - val_accuracy: 0.5524
Epoch 16/20
75/75 [==============================] - 1s 18ms/step - loss: 0.0046 - accur
acy: 1.0000 - val_loss: 1.8122 - val_accuracy: 0.5508
Epoch 17/20
75/75 [==============================] - 1s 18ms/step - loss: 0.0039 - accur
acy: 1.0000 - val_loss: 1.8252 - val_accuracy: 0.5576
Epoch 18/20
75/75 [==============================] - 2s 20ms/step - loss: 0.0034 - accur
acy: 1.0000 - val_loss: 1.8528 - val_accuracy: 0.5552
Epoch 19/20
75/75 [==============================] - 2s 25ms/step - loss: 0.0032 - accur
acy: 1.0000 - val_loss: 1.8693 - val_accuracy: 0.5512
Epoch 20/20
75/75 [==============================] - 2s 22ms/step - loss: 0.0026 - accur
acy: 1.0000 - val_loss: 1.8806 - val_accuracy: 0.5452
```

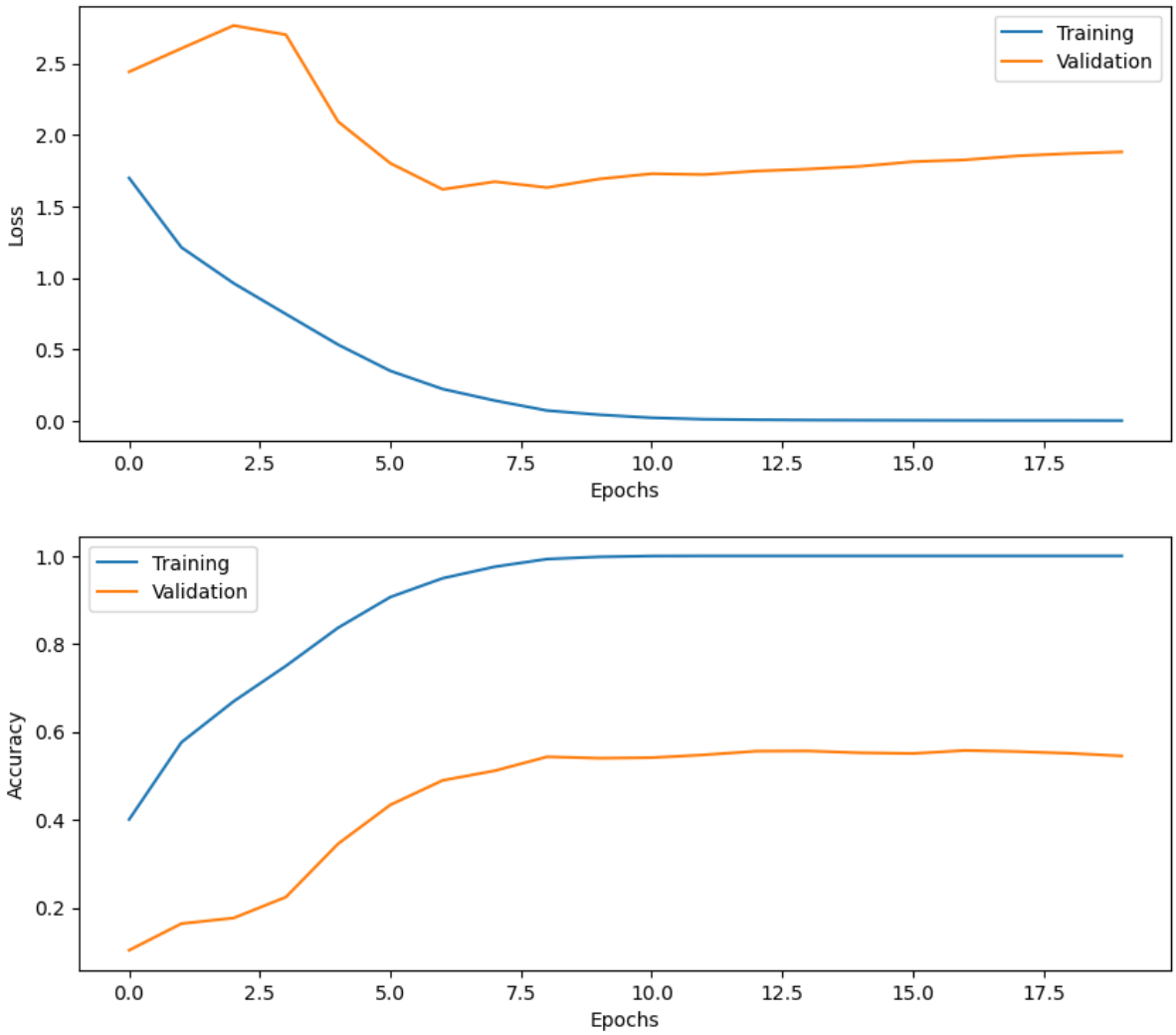
In [20]:

```
# Evaluate the trained model on test set, not used in training or validation
score = model3.evaluate(Xtest,Ytest)
```

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

```
63/63 [==============================] - 0s 3ms/step - loss: 1.8587 - accura
cy: 0.5630
Test loss: 1.8587
Test accuracy: 0.5630
```

In [21]:
```
# 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?

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

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

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?

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)?

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

Answer :

- A12 : 124,180 trainable parameters. The last Conv2D layer contains the most number of parameters.
- A13 : Input to the Conv2D is a 4-D tensor of dimensions (batch_size,rows, cols, channels) if data_format='channels_last'. Output of the Conv2D layer is 4-D tensor with dimensions: (batch_size,new_rows, new_cols,filters) if data_format='channels_last'.
- A14 : True,the batch_size is always the 1st dimension of the 4-D Tensor.
- A15 : Number of channels in the output : 128
- A16 : This is because the parameters are shared in a convolutional layer.
- A17 : Maxpooling outputs only the maximum value observed in a particular region. As such, it downsamples and reduces the number of parameters that are available to train.

In [22]:
```
# Print network architecture

model3.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_4 (Conv2D)           (None, 32, 32, 16)        448

 batch_normalization_5 (Batc  (None, 32, 32, 16)       64
 hNormalization)

 max_pooling2d_4 (MaxPooling  (None, 16, 16, 16)       0
 2D)

 conv2d_5 (Conv2D)           (None, 16, 16, 32)        4640

 batch_normalization_6 (Batc  (None, 16, 16, 32)       128
 hNormalization)

 max_pooling2d_5 (MaxPooling  (None, 8, 8, 32)         0
 2D)

 conv2d_6 (Conv2D)           (None, 8, 8, 64)          18496

 batch_normalization_7 (Batc  (None, 8, 8, 64)         256
 hNormalization)

 max_pooling2d_6 (MaxPooling  (None, 4, 4, 64)         0
 2D)

 conv2d_7 (Conv2D)           (None, 4, 4, 128)         73856

 batch_normalization_8 (Batc  (None, 4, 4, 128)        512
 hNormalization)

 max_pooling2d_7 (MaxPooling  (None, 2, 2, 128)        0
 2D)

 flatten_2 (Flatten)         (None, 512)               0

 dense_3 (Dense)             (None, 50)                25650

 batch_normalization_9 (Batc  (None, 50)               200
 hNormalization)

 dense_4 (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?

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

Answer

- A18 : The test accuracy of the model with dropout was 57.20% while the model without dropout achieved an test accuracy of 55.55%. An improvement of 1.65% is observed.

  However, sometimes we observed the accuracy being worse with dropout, so the improvement is not consistent
- A19 : Other types of regularization are L1 (Lasso) and L2 regularization. we can add L2 regularization for the convolutional layers by using "kernel_regularizer" argument when defining the Conv2D layer and passing an l2 object with an initialized value after importing it from keras.

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

In [23]:
```
# 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_filters=16,n_dense_layers=1

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

```
Epoch 1/20
75/75 [==============================] - 2s 22ms/step - loss: 2.3082 - accur
acy: 0.2783 - val_loss: 2.4557 - val_accuracy: 0.1072
Epoch 2/20
75/75 [==============================] - 1s 19ms/step - loss: 1.7015 - accur
acy: 0.4115 - val_loss: 2.5005 - val_accuracy: 0.1548
Epoch 3/20
75/75 [==============================] - 2s 21ms/step - loss: 1.4696 - accur
acy: 0.4728 - val_loss: 2.4288 - val_accuracy: 0.1816
Epoch 4/20
75/75 [==============================] - 2s 27ms/step - loss: 1.2934 - accur
acy: 0.5373 - val_loss: 2.2050 - val_accuracy: 0.2564
Epoch 5/20
75/75 [==============================] - 1s 19ms/step - loss: 1.1490 - accur
acy: 0.5945 - val_loss: 1.8120 - val_accuracy: 0.3568
Epoch 6/20
75/75 [==============================] - 2s 22ms/step - loss: 1.0263 - accur
acy: 0.6343 - val_loss: 1.4787 - val_accuracy: 0.4772
Epoch 7/20
75/75 [==============================] - 1s 18ms/step - loss: 0.9143 - accur
acy: 0.6839 - val_loss: 1.3999 - val_accuracy: 0.5088
Epoch 8/20
75/75 [==============================] - 1s 18ms/step - loss: 0.8012 - accur
acy: 0.7235 - val_loss: 1.4389 - val_accuracy: 0.5040
Epoch 9/20
75/75 [==============================] - 2s 21ms/step - loss: 0.6956 - accur
acy: 0.7599 - val_loss: 1.3178 - val_accuracy: 0.5428
Epoch 10/20
75/75 [==============================] - 2s 21ms/step - loss: 0.6022 - accur
acy: 0.7907 - val_loss: 1.3741 - val_accuracy: 0.5432
Epoch 11/20
75/75 [==============================] - 1s 19ms/step - loss: 0.5105 - accur
acy: 0.8336 - val_loss: 1.4963 - val_accuracy: 0.5392
Epoch 12/20
75/75 [==============================] - 1s 20ms/step - loss: 0.4657 - accur
acy: 0.8451 - val_loss: 1.4727 - val_accuracy: 0.5420
Epoch 13/20
75/75 [==============================] - 1s 20ms/step - loss: 0.3872 - accur
acy: 0.8721 - val_loss: 1.4775 - val_accuracy: 0.5540
Epoch 14/20
75/75 [==============================] - 1s 18ms/step - loss: 0.3283 - accur
acy: 0.8996 - val_loss: 1.7047 - val_accuracy: 0.5316
Epoch 15/20
75/75 [==============================] - 1s 18ms/step - loss: 0.2877 - accur
acy: 0.9065 - val_loss: 1.6884 - val_accuracy: 0.5340
Epoch 16/20
75/75 [==============================] - 2s 20ms/step - loss: 0.2396 - accur
acy: 0.9265 - val_loss: 1.5967 - val_accuracy: 0.5568
Epoch 17/20
75/75 [==============================] - 1s 19ms/step - loss: 0.2030 - accur
acy: 0.9392 - val_loss: 1.7316 - val_accuracy: 0.5504
Epoch 18/20
75/75 [==============================] - 1s 18ms/step - loss: 0.1750 - accur
acy: 0.9484 - val_loss: 1.7853 - val_accuracy: 0.5512
Epoch 19/20
75/75 [==============================] - 1s 20ms/step - loss: 0.1518 - accur
acy: 0.9555 - val_loss: 1.8173 - val_accuracy: 0.5580
Epoch 20/20
75/75 [==============================] - 1s 20ms/step - loss: 0.1437 - accur
acy: 0.9573 - val_loss: 1.8933 - val_accuracy: 0.5540
```

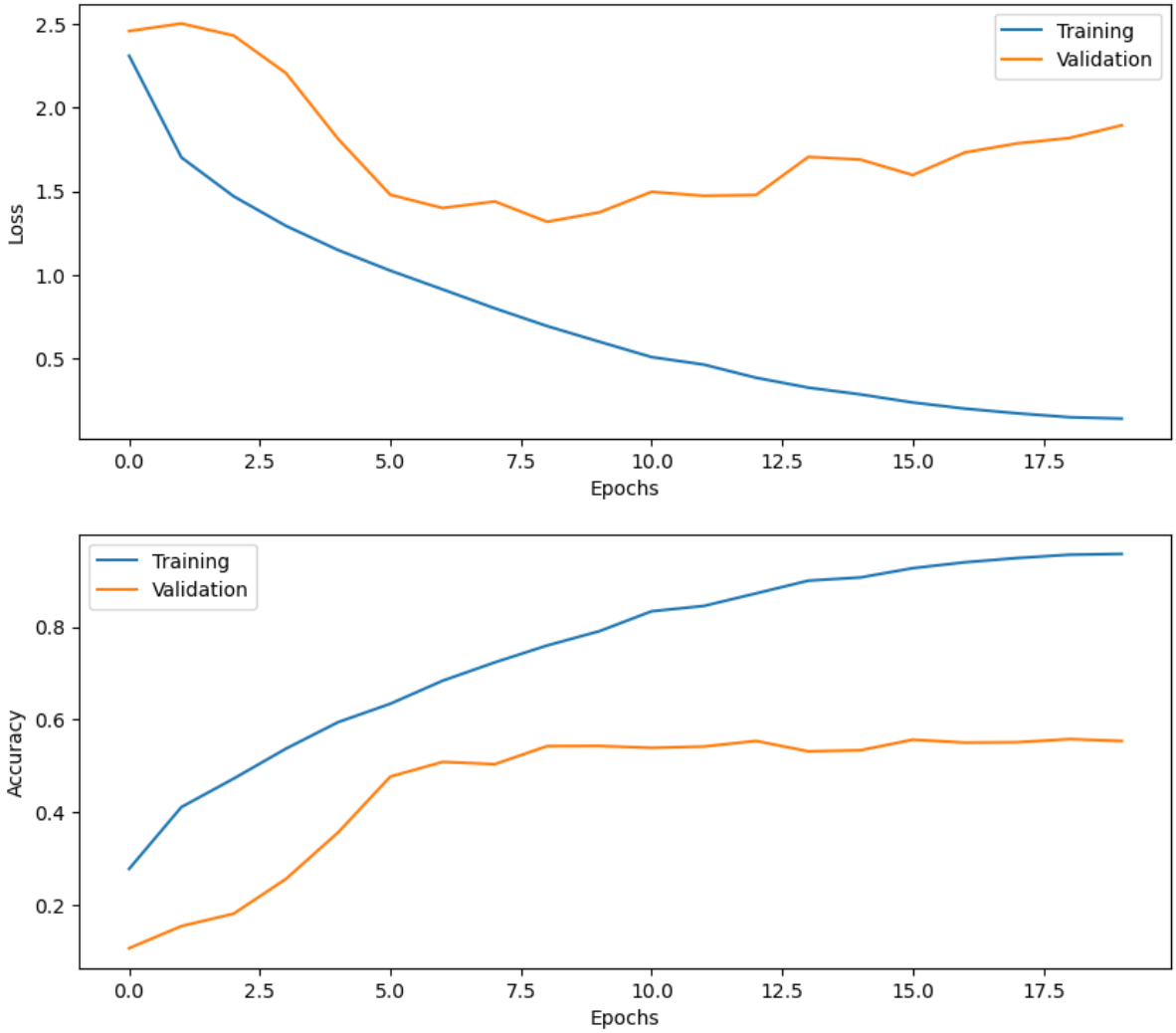
In [24]:

```
# Evaluate the trained model on test set, not used in training or validation
score = model4.evaluate(Xtest,Ytest)
```

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

```
63/63 [==============================] - 0s 2ms/step - loss: 1.7876 - accura
cy: 0.5625
Test loss: 1.7876
Test accuracy: 0.5625
```

In [25]:
```
# Plot the history from the training run
plot_results(history4)
```



Part 15: Tweaking performance

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

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

Answer :

- A20 : Highest accuracy achieved is 68.65% (mostly around 66.5%).

- Configuration :
- Convolution Layers : 4
- Number of filters : 64
- Number of Dense layers : 2
- Number of nodes : 64
- Dropout : True
- Learning Rate : 0.1
- Batch size : 150
- epochs : 25

Your best config

In [41]:
```
# Setup some training parameters
batch_size = 150
epochs = 25
input_shape = Xtrain.shape[1:]

# Build model
model5 = build_CNN(input_shape,n_conv_layers=4,n_filters=64,n_dense_layers=2

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

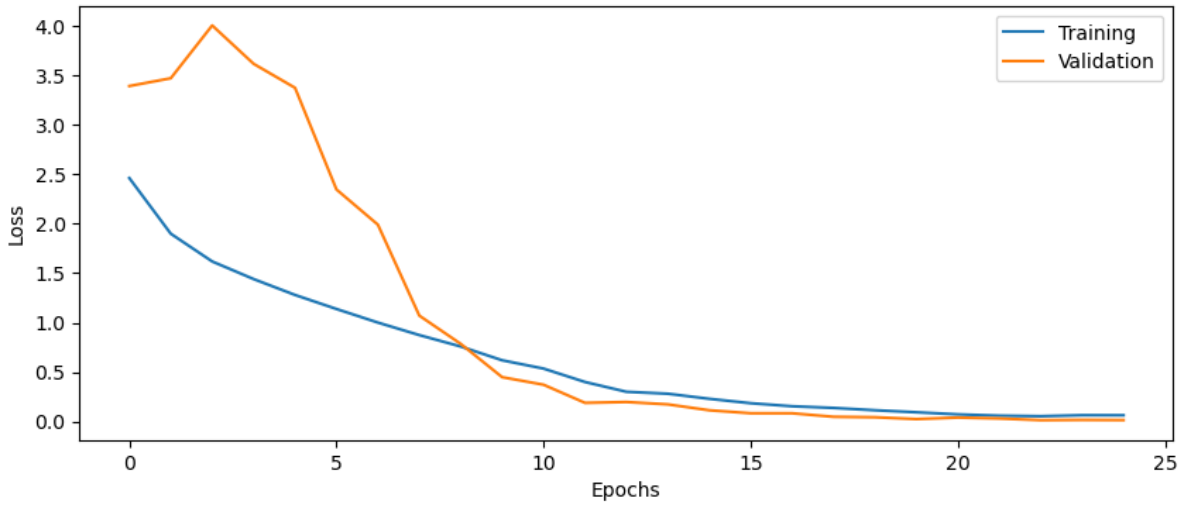
```
Epoch 1/25
67/67 [==============================] - 12s 161ms/step - loss: 2.4621 - acc
uracy: 0.2364 - val_loss: 3.3932 - val_accuracy: 0.1108
Epoch 2/25
67/67 [==============================] - 10s 154ms/step - loss: 1.8998 - acc
uracy: 0.3454 - val_loss: 3.4717 - val_accuracy: 0.0996
Epoch 3/25
67/67 [==============================] - 11s 158ms/step - loss: 1.6191 - acc
uracy: 0.4124 - val_loss: 4.0060 - val_accuracy: 0.0996
Epoch 4/25
67/67 [==============================] - 11s 158ms/step - loss: 1.4410 - acc
uracy: 0.4788 - val_loss: 3.6179 - val_accuracy: 0.1008
Epoch 5/25
67/67 [==============================] - 11s 159ms/step - loss: 1.2804 - acc
uracy: 0.5344 - val_loss: 3.3750 - val_accuracy: 0.1336
Epoch 6/25
67/67 [==============================] - 11s 160ms/step - loss: 1.1384 - acc
uracy: 0.5958 - val_loss: 2.3470 - val_accuracy: 0.2284
Epoch 7/25
67/67 [==============================] - 11s 159ms/step - loss: 1.0011 - acc
uracy: 0.6463 - val_loss: 1.9911 - val_accuracy: 0.3884
Epoch 8/25
67/67 [==============================] - 11s 169ms/step - loss: 0.8745 - acc
uracy: 0.6979 - val_loss: 1.0714 - val_accuracy: 0.5924
Epoch 9/25
67/67 [==============================] - 11s 170ms/step - loss: 0.7589 - acc
uracy: 0.7416 - val_loss: 0.7856 - val_accuracy: 0.7204
Epoch 10/25
67/67 [==============================] - 11s 170ms/step - loss: 0.6196 - acc
uracy: 0.7931 - val_loss: 0.4488 - val_accuracy: 0.8552
Epoch 11/25
67/67 [==============================] - 11s 168ms/step - loss: 0.5356 - acc
uracy: 0.8259 - val_loss: 0.3724 - val_accuracy: 0.8716
Epoch 12/25
67/67 [==============================] - 11s 166ms/step - loss: 0.4000 - acc
uracy: 0.8731 - val_loss: 0.1891 - val_accuracy: 0.9472
Epoch 13/25
67/67 [==============================] - 11s 167ms/step - loss: 0.3010 - acc
uracy: 0.9109 - val_loss: 0.1976 - val_accuracy: 0.9392
Epoch 14/25
67/67 [==============================] - 11s 165ms/step - loss: 0.2815 - acc
uracy: 0.9155 - val_loss: 0.1738 - val_accuracy: 0.9496
Epoch 15/25
67/67 [==============================] - 11s 169ms/step - loss: 0.2300 - acc
uracy: 0.9307 - val_loss: 0.1141 - val_accuracy: 0.9632
Epoch 16/25
67/67 [==============================] - 11s 167ms/step - loss: 0.1854 - acc
uracy: 0.9442 - val_loss: 0.0840 - val_accuracy: 0.9736
Epoch 17/25
67/67 [==============================] - 12s 172ms/step - loss: 0.1547 - acc
uracy: 0.9559 - val_loss: 0.0836 - val_accuracy: 0.9740
Epoch 18/25
67/67 [==============================] - 11s 165ms/step - loss: 0.1378 - acc
uracy: 0.9598 - val_loss: 0.0481 - val_accuracy: 0.9852
Epoch 19/25
67/67 [==============================] - 11s 160ms/step - loss: 0.1144 - acc
uracy: 0.9677 - val_loss: 0.0435 - val_accuracy: 0.9856
Epoch 20/25
67/67 [==============================] - 11s 158ms/step - loss: 0.0938 - acc
uracy: 0.9749 - val_loss: 0.0250 - val_accuracy: 0.9920
Epoch 21/25
67/67 [==============================] - 11s 157ms/step - loss: 0.0726 - acc
uracy: 0.9799 - val_loss: 0.0403 - val_accuracy: 0.9892
Epoch 22/25
```

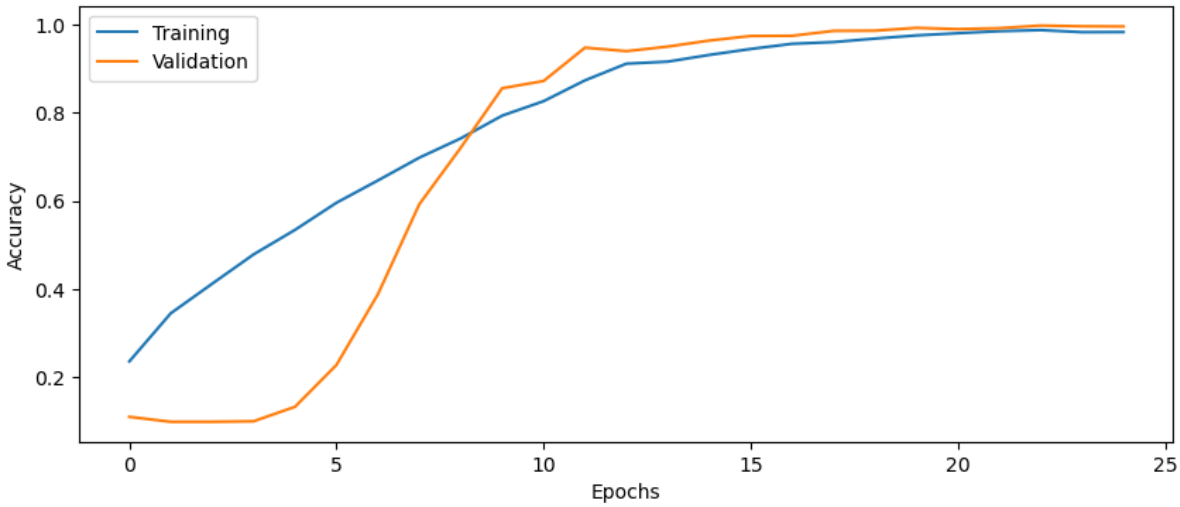
```
67/67 [==============================] - 11s 158ms/step - loss: 0.0594 - acc
uracy: 0.9845 - val_loss: 0.0309 - val_accuracy: 0.9912
Epoch 23/25
67/67 [==============================] - 11s 163ms/step - loss: 0.0542 - acc
uracy: 0.9869 - val_loss: 0.0141 - val_accuracy: 0.9972
Epoch 24/25
67/67 [==============================] - 11s 162ms/step - loss: 0.0642 - acc
uracy: 0.9823 - val_loss: 0.0166 - val_accuracy: 0.9956
Epoch 25/25
67/67 [==============================] - 11s 162ms/step - loss: 0.0638 - acc
uracy: 0.9826 - val_loss: 0.0148 - val_accuracy: 0.9952
```

In [42]:
```
# Evaluate the trained model on test set, not used in training or validation
score = model5.evaluate(Xtest,Ytest)
print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])
```

```
63/63 [==============================] - 1s 16ms/step - loss: 1.4290 - accur
acy: 0.6865
Test loss: 1.4290
Test accuracy: 0.6865
```

In [43]:
```
# Plot the history from the training run
plot_results(history5)
```





Part 16: Rotate the test images

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

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

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

Answer :

- A21 : The test accuracy for the rotated models is around 24% compared to our best accuracy around 67%. This is because the model was trained on non-rotated images. When the images in the test set are rotated, the pixels' positions are also changed,meaning the outputs from the filters are also different compared to the outputs from the non-rotated training data, and the neural network was not trained in the aspect. This causes the model to misclassify images.

In [29]:
```
def myrotate(images):

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

    return images_rot
```

In [30]:
```
# 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 [53]:
```
# Evaluate the trained model on rotated test set
score = model5.evaluate(Xtest_rotated,Ytest)
print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])
```

```
63/63 [==============================] - 1s 16ms/step - loss: 4.4472 - accur
acy: 0.2605
Test loss: 4.4472
Test accuracy: 0.2605
```

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/ImageDataGenera
, 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 [32]:
```
# Get all 60 000 training images again. ImageDataGenerator manages validatic
(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 [56]:
```
# Set up a data generator with on-the-fly data augmentation, 20% validation
# Use a rotation range of 30 degrees, horizontal and vertical flipping
from keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(rotation_range=30,horizontal_flip=True,vertical

# Setup a flow for training data, assume that we can fit all images into CPU
# Setup a flow for validation data, assume that we can fit all images into C

Train,Valid = datagen.flow(Xtrain, Ytrain, batch_size=batch_size,subset='tra


#Train = datagen.fit(Xtrain)
# fits the model on batches with real-time data augmentation:
```

Part 18: What about big data?

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

Answer :

- A22 : We would load the training data into memory in batches and proceed from there. This can be done using the flow_from_directory function from tensorflow.keras.preprocessing.image. The disadvantage is that the trainng and augmentation time will increase.

In [57]:
```
# 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?

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

Answer :

- A23 : Without Augmentations, it takes 7 epochs taken to reach 60% accuracy, whereas with augmentation it takes 21 epochs to reach 60% accuracy. This is because in the case with augmentation, we are also rotating the images and flipping the training images horizontally and vertically. These are big changes which increase the complexity of the problem which in turn, slows down the training rate or speed.
- A24 : Other types of image augmentation include rescaling,height and width shifting ,brightness tuning.

In [59]:
```
# Setup some training parameters
batch_size = 100
epochs = 200
input_shape = Xtrain.shape[1:4]

# Build model (your best config)
model6 = build_CNN(input_shape,n_conv_layers=4,n_filters=64,n_dense_layers=2

validation_split=0.2
steps_per_epoch=len(Xtrain)*(1 - validation_split)/batch_size
validation_steps= len(Xtrain)*validation_split/batch_size
# Train the model using on the fly augmentation
history6 = model6.fit(Train, batch_size = batch_size,epochs = epochs,validat
                      steps_per_epoch = steps_per_epoch,
                      validation_steps= validation_steps)
```

```
Epoch 1/200
80/80 [==============================] - 11s 124ms/step - loss: 2.6611 - acc
uracy: 0.1834 - val_loss: 2.5562 - val_accuracy: 0.1040
Epoch 2/200
80/80 [==============================] - 11s 133ms/step - loss: 2.1581 - acc
uracy: 0.2660 - val_loss: 3.9850 - val_accuracy: 0.1015
Epoch 3/200
80/80 [==============================] - 10s 126ms/step - loss: 1.9364 - acc
uracy: 0.3070 - val_loss: 3.6765 - val_accuracy: 0.1025
Epoch 4/200
80/80 [==============================] - 10s 119ms/step - loss: 1.8424 - acc
uracy: 0.3288 - val_loss: 2.9371 - val_accuracy: 0.1620
Epoch 5/200
80/80 [==============================] - 12s 147ms/step - loss: 1.7373 - acc
uracy: 0.3574 - val_loss: 2.6505 - val_accuracy: 0.1955
Epoch 6/200
80/80 [==============================] - 11s 143ms/step - loss: 1.6628 - acc
uracy: 0.3881 - val_loss: 1.9444 - val_accuracy: 0.2760
Epoch 7/200
80/80 [==============================] - 12s 145ms/step - loss: 1.5941 - acc
uracy: 0.4034 - val_loss: 1.6834 - val_accuracy: 0.3865
Epoch 8/200
80/80 [==============================] - 11s 135ms/step - loss: 1.5602 - acc
uracy: 0.4214 - val_loss: 1.4101 - val_accuracy: 0.4740
Epoch 9/200
80/80 [==============================] - 10s 123ms/step - loss: 1.5041 - acc
uracy: 0.4431 - val_loss: 1.3680 - val_accuracy: 0.5010
Epoch 10/200
80/80 [==============================] - 9s 113ms/step - loss: 1.4922 - accu
racy: 0.4539 - val_loss: 1.3525 - val_accuracy: 0.5055
Epoch 11/200
80/80 [==============================] - 10s 118ms/step - loss: 1.4252 - acc
uracy: 0.4800 - val_loss: 1.3840 - val_accuracy: 0.4965
Epoch 12/200
80/80 [==============================] - 9s 115ms/step - loss: 1.3897 - accu
racy: 0.4963 - val_loss: 1.3141 - val_accuracy: 0.5175
Epoch 13/200
80/80 [==============================] - 11s 139ms/step - loss: 1.3556 - acc
uracy: 0.5086 - val_loss: 1.2636 - val_accuracy: 0.5330
Epoch 14/200
80/80 [==============================] - 11s 141ms/step - loss: 1.3267 - acc
uracy: 0.5249 - val_loss: 1.2571 - val_accuracy: 0.5370
Epoch 15/200
80/80 [==============================] - 12s 147ms/step - loss: 1.2911 - acc
uracy: 0.5480 - val_loss: 1.2237 - val_accuracy: 0.5580
Epoch 16/200
80/80 [==============================] - 10s 130ms/step - loss: 1.2587 - acc
uracy: 0.5495 - val_loss: 1.2647 - val_accuracy: 0.5505
Epoch 17/200
80/80 [==============================] - 10s 122ms/step - loss: 1.2355 - acc
uracy: 0.5589 - val_loss: 1.1733 - val_accuracy: 0.5665
Epoch 18/200
80/80 [==============================] - 9s 112ms/step - loss: 1.2236 - accu
racy: 0.5663 - val_loss: 1.1365 - val_accuracy: 0.5935
Epoch 19/200
80/80 [==============================] - 11s 133ms/step - loss: 1.2133 - acc
uracy: 0.5729 - val_loss: 1.2328 - val_accuracy: 0.5680
Epoch 20/200
80/80 [==============================] - 12s 144ms/step - loss: 1.1810 - acc
uracy: 0.5956 - val_loss: 1.1822 - val_accuracy: 0.5855
Epoch 21/200
80/80 [==============================] - 10s 126ms/step - loss: 1.1271 - acc
uracy: 0.6094 - val_loss: 1.1495 - val_accuracy: 0.5965
Epoch 22/200
```

```
80/80 [==============================] - 9s 118ms/step - loss: 1.1184 - accu
racy: 0.6094 - val_loss: 1.2091 - val_accuracy: 0.5705
Epoch 23/200
80/80 [==============================] - 10s 129ms/step - loss: 1.0833 - acc
uracy: 0.6173 - val_loss: 1.3058 - val_accuracy: 0.5415
Epoch 24/200
80/80 [==============================] - 10s 124ms/step - loss: 1.0620 - acc
uracy: 0.6339 - val_loss: 1.0799 - val_accuracy: 0.6140
Epoch 25/200
80/80 [==============================] - 11s 132ms/step - loss: 1.0294 - acc
uracy: 0.6404 - val_loss: 1.1272 - val_accuracy: 0.6170
Epoch 26/200
80/80 [==============================] - 9s 116ms/step - loss: 1.0234 - accu
racy: 0.6415 - val_loss: 1.1305 - val_accuracy: 0.5950
Epoch 27/200
80/80 [==============================] - 10s 119ms/step - loss: 1.0102 - acc
uracy: 0.6495 - val_loss: 1.0898 - val_accuracy: 0.6140
Epoch 28/200
80/80 [==============================] - 10s 125ms/step - loss: 0.9965 - acc
uracy: 0.6611 - val_loss: 1.0538 - val_accuracy: 0.6270
Epoch 29/200
80/80 [==============================] - 9s 118ms/step - loss: 0.9609 - accu
racy: 0.6658 - val_loss: 1.0332 - val_accuracy: 0.6375
Epoch 30/200
80/80 [==============================] - 9s 113ms/step - loss: 0.9534 - accu
racy: 0.6766 - val_loss: 1.1188 - val_accuracy: 0.6160
Epoch 31/200
80/80 [==============================] - 10s 120ms/step - loss: 0.9275 - acc
uracy: 0.6819 - val_loss: 1.1533 - val_accuracy: 0.5855
Epoch 32/200
80/80 [==============================] - 11s 133ms/step - loss: 0.9161 - acc
uracy: 0.6886 - val_loss: 1.0505 - val_accuracy: 0.6295
Epoch 33/200
80/80 [==============================] - 10s 120ms/step - loss: 0.8899 - acc
uracy: 0.6942 - val_loss: 1.0719 - val_accuracy: 0.6240
Epoch 34/200
80/80 [==============================] - 10s 122ms/step - loss: 0.8797 - acc
uracy: 0.7014 - val_loss: 1.2093 - val_accuracy: 0.6085
Epoch 35/200
80/80 [==============================] - 10s 128ms/step - loss: 0.8541 - acc
uracy: 0.7104 - val_loss: 1.0901 - val_accuracy: 0.6235
Epoch 36/200
80/80 [==============================] - 9s 114ms/step - loss: 0.8574 - accu
racy: 0.7101 - val_loss: 1.0543 - val_accuracy: 0.6390
Epoch 37/200
80/80 [==============================] - 10s 126ms/step - loss: 0.8401 - acc
uracy: 0.7129 - val_loss: 1.0333 - val_accuracy: 0.6550
Epoch 38/200
80/80 [==============================] - 10s 127ms/step - loss: 0.8195 - acc
uracy: 0.7235 - val_loss: 1.0982 - val_accuracy: 0.6360
Epoch 39/200
80/80 [==============================] - 9s 114ms/step - loss: 0.8097 - accu
racy: 0.7297 - val_loss: 1.0747 - val_accuracy: 0.6480
Epoch 40/200
80/80 [==============================] - 10s 120ms/step - loss: 0.7817 - acc
uracy: 0.7374 - val_loss: 1.0836 - val_accuracy: 0.6275
Epoch 41/200
80/80 [==============================] - 10s 125ms/step - loss: 0.7622 - acc
uracy: 0.7424 - val_loss: 1.0736 - val_accuracy: 0.6535
Epoch 42/200
80/80 [==============================] - 10s 129ms/step - loss: 0.7384 - acc
uracy: 0.7475 - val_loss: 1.0693 - val_accuracy: 0.6385
Epoch 43/200
80/80 [==============================] - 10s 127ms/step - loss: 0.7273 - acc
```

```
uracy: 0.7566 - val_loss: 1.0390 - val_accuracy: 0.6535
Epoch 44/200
80/80 [==============================] - 9s 115ms/step - loss: 0.7247 - accu
racy: 0.7596 - val_loss: 1.1093 - val_accuracy: 0.6420
Epoch 45/200
80/80 [==============================] - 9s 115ms/step - loss: 0.7182 - accu
racy: 0.7623 - val_loss: 1.0487 - val_accuracy: 0.6510
Epoch 46/200
80/80 [==============================] - 9s 117ms/step - loss: 0.6822 - accu
racy: 0.7741 - val_loss: 1.1147 - val_accuracy: 0.6395
Epoch 47/200
80/80 [==============================] - 10s 130ms/step - loss: 0.6745 - acc
uracy: 0.7756 - val_loss: 1.0662 - val_accuracy: 0.6575
Epoch 48/200
80/80 [==============================] - 10s 121ms/step - loss: 0.6815 - acc
uracy: 0.7761 - val_loss: 1.0710 - val_accuracy: 0.6625
Epoch 49/200
80/80 [==============================] - 10s 128ms/step - loss: 0.6543 - acc
uracy: 0.7821 - val_loss: 1.1132 - val_accuracy: 0.6570
Epoch 50/200
80/80 [==============================] - 10s 121ms/step - loss: 0.6470 - acc
uracy: 0.7897 - val_loss: 1.0077 - val_accuracy: 0.6705
Epoch 51/200
80/80 [==============================] - 10s 130ms/step - loss: 0.6342 - acc
uracy: 0.7919 - val_loss: 1.0675 - val_accuracy: 0.6640
Epoch 52/200
80/80 [==============================] - 9s 116ms/step - loss: 0.6249 - accu
racy: 0.7904 - val_loss: 0.9961 - val_accuracy: 0.6840
Epoch 53/200
80/80 [==============================] - 10s 121ms/step - loss: 0.6155 - acc
uracy: 0.7979 - val_loss: 1.0483 - val_accuracy: 0.6670
Epoch 54/200
80/80 [==============================] - 10s 121ms/step - loss: 0.5966 - acc
uracy: 0.8023 - val_loss: 1.0821 - val_accuracy: 0.6615
Epoch 55/200
80/80 [==============================] - 10s 121ms/step - loss: 0.5974 - acc
uracy: 0.8051 - val_loss: 1.0574 - val_accuracy: 0.6665
Epoch 56/200
80/80 [==============================] - 10s 122ms/step - loss: 0.5831 - acc
uracy: 0.8065 - val_loss: 1.0927 - val_accuracy: 0.6580
Epoch 57/200
80/80 [==============================] - 9s 117ms/step - loss: 0.5624 - accu
racy: 0.8151 - val_loss: 1.1085 - val_accuracy: 0.6635
Epoch 58/200
80/80 [==============================] - 9s 117ms/step - loss: 0.5533 - accu
racy: 0.8216 - val_loss: 1.0579 - val_accuracy: 0.6700
Epoch 59/200
80/80 [==============================] - 10s 126ms/step - loss: 0.5447 - acc
uracy: 0.8191 - val_loss: 1.1061 - val_accuracy: 0.6590
Epoch 60/200
80/80 [==============================] - 9s 117ms/step - loss: 0.5259 - accu
racy: 0.8325 - val_loss: 1.0401 - val_accuracy: 0.6875
Epoch 61/200
80/80 [==============================] - 9s 117ms/step - loss: 0.5281 - accu
racy: 0.8304 - val_loss: 1.2253 - val_accuracy: 0.6440
Epoch 62/200
80/80 [==============================] - 10s 125ms/step - loss: 0.5310 - acc
uracy: 0.8304 - val_loss: 1.1957 - val_accuracy: 0.6575
Epoch 63/200
80/80 [==============================] - 11s 132ms/step - loss: 0.5385 - acc
uracy: 0.8265 - val_loss: 1.1171 - val_accuracy: 0.6750
Epoch 64/200
80/80 [==============================] - 11s 140ms/step - loss: 0.4902 - acc
uracy: 0.8367 - val_loss: 1.1052 - val_accuracy: 0.6630
```

```
Epoch 65/200
80/80 [==============================] - 11s 134ms/step - loss: 0.4875 - acc
uracy: 0.8440 - val_loss: 1.0729 - val_accuracy: 0.6845
Epoch 66/200
80/80 [==============================] - 9s 116ms/step - loss: 0.4953 - accu
racy: 0.8394 - val_loss: 1.1410 - val_accuracy: 0.6670
Epoch 67/200
80/80 [==============================] - 12s 145ms/step - loss: 0.4743 - acc
uracy: 0.8469 - val_loss: 1.1120 - val_accuracy: 0.6675
Epoch 68/200
80/80 [==============================] - 10s 128ms/step - loss: 0.4650 - acc
uracy: 0.8529 - val_loss: 1.1545 - val_accuracy: 0.6620
Epoch 69/200
80/80 [==============================] - 9s 116ms/step - loss: 0.4569 - accu
racy: 0.8533 - val_loss: 1.3406 - val_accuracy: 0.6330
Epoch 70/200
80/80 [==============================] - 10s 125ms/step - loss: 0.4450 - acc
uracy: 0.8536 - val_loss: 1.1273 - val_accuracy: 0.6770
Epoch 71/200
80/80 [==============================] - 10s 125ms/step - loss: 0.4489 - acc
uracy: 0.8505 - val_loss: 1.1022 - val_accuracy: 0.6805
Epoch 72/200
80/80 [==============================] - 9s 117ms/step - loss: 0.4325 - accu
racy: 0.8622 - val_loss: 1.1463 - val_accuracy: 0.6670
Epoch 73/200
80/80 [==============================] - 10s 122ms/step - loss: 0.4234 - acc
uracy: 0.8575 - val_loss: 1.1640 - val_accuracy: 0.6710
Epoch 74/200
80/80 [==============================] - 10s 128ms/step - loss: 0.4421 - acc
uracy: 0.8585 - val_loss: 1.1597 - val_accuracy: 0.6845
Epoch 75/200
80/80 [==============================] - 10s 119ms/step - loss: 0.4214 - acc
uracy: 0.8661 - val_loss: 1.1539 - val_accuracy: 0.6735
Epoch 76/200
80/80 [==============================] - 9s 117ms/step - loss: 0.4175 - accu
racy: 0.8631 - val_loss: 1.1756 - val_accuracy: 0.6735
Epoch 77/200
80/80 [==============================] - 10s 123ms/step - loss: 0.4145 - acc
uracy: 0.8687 - val_loss: 1.1977 - val_accuracy: 0.6685
Epoch 78/200
80/80 [==============================] - 9s 116ms/step - loss: 0.4000 - accu
racy: 0.8749 - val_loss: 1.1227 - val_accuracy: 0.6815
Epoch 79/200
80/80 [==============================] - 10s 123ms/step - loss: 0.3943 - acc
uracy: 0.8731 - val_loss: 1.1252 - val_accuracy: 0.6910
Epoch 80/200
80/80 [==============================] - 10s 126ms/step - loss: 0.3808 - acc
uracy: 0.8763 - val_loss: 1.2021 - val_accuracy: 0.6705
Epoch 81/200
80/80 [==============================] - 12s 149ms/step - loss: 0.3680 - acc
uracy: 0.8796 - val_loss: 1.1424 - val_accuracy: 0.6885
Epoch 82/200
80/80 [==============================] - 9s 117ms/step - loss: 0.3617 - accu
racy: 0.8863 - val_loss: 1.1669 - val_accuracy: 0.6765
Epoch 83/200
80/80 [==============================] - 9s 118ms/step - loss: 0.3671 - accu
racy: 0.8848 - val_loss: 1.1579 - val_accuracy: 0.6840
Epoch 84/200
80/80 [==============================] - 10s 127ms/step - loss: 0.3424 - acc
uracy: 0.8926 - val_loss: 1.1937 - val_accuracy: 0.6835
Epoch 85/200
80/80 [==============================] - 10s 121ms/step - loss: 0.3396 - acc
uracy: 0.8882 - val_loss: 1.2379 - val_accuracy: 0.6725
Epoch 86/200
```

```
80/80 [==============================] - 12s 150ms/step - loss: 0.3463 - acc
uracy: 0.8892 - val_loss: 1.1991 - val_accuracy: 0.6805
Epoch 87/200
80/80 [==============================] - 10s 125ms/step - loss: 0.3376 - acc
uracy: 0.8915 - val_loss: 1.2249 - val_accuracy: 0.6710
Epoch 88/200
80/80 [==============================] - 9s 118ms/step - loss: 0.3447 - accu
racy: 0.8928 - val_loss: 1.1729 - val_accuracy: 0.6865
Epoch 89/200
80/80 [==============================] - 10s 121ms/step - loss: 0.3353 - acc
uracy: 0.8905 - val_loss: 1.1997 - val_accuracy: 0.6760
Epoch 90/200
80/80 [==============================] - 9s 116ms/step - loss: 0.3468 - accu
racy: 0.8878 - val_loss: 1.1454 - val_accuracy: 0.7015
Epoch 91/200
80/80 [==============================] - 10s 122ms/step - loss: 0.3254 - acc
uracy: 0.8996 - val_loss: 1.1924 - val_accuracy: 0.6885
Epoch 92/200
80/80 [==============================] - 10s 121ms/step - loss: 0.3213 - acc
uracy: 0.8981 - val_loss: 1.1997 - val_accuracy: 0.6800
Epoch 93/200
80/80 [==============================] - 10s 122ms/step - loss: 0.3187 - acc
uracy: 0.9011 - val_loss: 1.1602 - val_accuracy: 0.6960
Epoch 94/200
80/80 [==============================] - 9s 114ms/step - loss: 0.2876 - accu
racy: 0.9101 - val_loss: 1.2250 - val_accuracy: 0.6825
Epoch 95/200
80/80 [==============================] - 10s 125ms/step - loss: 0.3174 - acc
uracy: 0.8980 - val_loss: 1.2387 - val_accuracy: 0.6960
Epoch 96/200
80/80 [==============================] - 10s 119ms/step - loss: 0.3099 - acc
uracy: 0.9010 - val_loss: 1.2569 - val_accuracy: 0.6775
Epoch 97/200
80/80 [==============================] - 9s 118ms/step - loss: 0.3199 - accu
racy: 0.8990 - val_loss: 1.2567 - val_accuracy: 0.6715
Epoch 98/200
80/80 [==============================] - 9s 117ms/step - loss: 0.3073 - accu
racy: 0.9038 - val_loss: 1.2702 - val_accuracy: 0.6640
Epoch 99/200
80/80 [==============================] - 10s 119ms/step - loss: 0.2815 - acc
uracy: 0.9111 - val_loss: 1.2709 - val_accuracy: 0.6775
Epoch 100/200
80/80 [==============================] - 9s 118ms/step - loss: 0.2982 - accu
racy: 0.9072 - val_loss: 1.2525 - val_accuracy: 0.6795
Epoch 101/200
80/80 [==============================] - 10s 123ms/step - loss: 0.2988 - acc
uracy: 0.9066 - val_loss: 1.2752 - val_accuracy: 0.6760
Epoch 102/200
80/80 [==============================] - 11s 138ms/step - loss: 0.2694 - acc
uracy: 0.9176 - val_loss: 1.3219 - val_accuracy: 0.6745
Epoch 103/200
80/80 [==============================] - 11s 143ms/step - loss: 0.2801 - acc
uracy: 0.9131 - val_loss: 1.2528 - val_accuracy: 0.6785
Epoch 104/200
80/80 [==============================] - 11s 131ms/step - loss: 0.2833 - acc
uracy: 0.9110 - val_loss: 1.3165 - val_accuracy: 0.6825
Epoch 105/200
80/80 [==============================] - 12s 148ms/step - loss: 0.2685 - acc
uracy: 0.9166 - val_loss: 1.2414 - val_accuracy: 0.6975
Epoch 106/200
80/80 [==============================] - 11s 134ms/step - loss: 0.2381 - acc
uracy: 0.9222 - val_loss: 1.3686 - val_accuracy: 0.6630
Epoch 107/200
80/80 [==============================] - 13s 162ms/step - loss: 0.2616 - acc
```

```
uracy: 0.9164 - val_loss: 1.3158 - val_accuracy: 0.6765
Epoch 108/200
80/80 [==============================] - 11s 143ms/step - loss: 0.2594 - acc
uracy: 0.9175 - val_loss: 1.2759 - val_accuracy: 0.6855
Epoch 109/200
80/80 [==============================] - 11s 133ms/step - loss: 0.2769 - acc
uracy: 0.9128 - val_loss: 1.3588 - val_accuracy: 0.6645
Epoch 110/200
80/80 [==============================] - 10s 120ms/step - loss: 0.2581 - acc
uracy: 0.9205 - val_loss: 1.3140 - val_accuracy: 0.6840
Epoch 111/200
80/80 [==============================] - 10s 119ms/step - loss: 0.2754 - acc
uracy: 0.9158 - val_loss: 1.2826 - val_accuracy: 0.6925
Epoch 112/200
80/80 [==============================] - 10s 122ms/step - loss: 0.2553 - acc
uracy: 0.9206 - val_loss: 1.3652 - val_accuracy: 0.6615
Epoch 113/200
80/80 [==============================] - 11s 133ms/step - loss: 0.2523 - acc
uracy: 0.9183 - val_loss: 1.2771 - val_accuracy: 0.6890
Epoch 114/200
80/80 [==============================] - 12s 154ms/step - loss: 0.2477 - acc
uracy: 0.9229 - val_loss: 1.3055 - val_accuracy: 0.6855
Epoch 115/200
80/80 [==============================] - 11s 134ms/step - loss: 0.2466 - acc
uracy: 0.9233 - val_loss: 1.2289 - val_accuracy: 0.6885
Epoch 116/200
80/80 [==============================] - 11s 132ms/step - loss: 0.2390 - acc
uracy: 0.9240 - val_loss: 1.3246 - val_accuracy: 0.6890
Epoch 117/200
80/80 [==============================] - 12s 151ms/step - loss: 0.2432 - acc
uracy: 0.9273 - val_loss: 1.2477 - val_accuracy: 0.6920
Epoch 118/200
80/80 [==============================] - 11s 135ms/step - loss: 0.2374 - acc
uracy: 0.9264 - val_loss: 1.2976 - val_accuracy: 0.6895
Epoch 119/200
80/80 [==============================] - 12s 152ms/step - loss: 0.2222 - acc
uracy: 0.9306 - val_loss: 1.3288 - val_accuracy: 0.6865
Epoch 120/200
80/80 [==============================] - 10s 122ms/step - loss: 0.2183 - acc
uracy: 0.9312 - val_loss: 1.3436 - val_accuracy: 0.6815
Epoch 121/200
80/80 [==============================] - 11s 134ms/step - loss: 0.2345 - acc
uracy: 0.9280 - val_loss: 1.3205 - val_accuracy: 0.6980
Epoch 122/200
80/80 [==============================] - 11s 141ms/step - loss: 0.2280 - acc
uracy: 0.9284 - val_loss: 1.3091 - val_accuracy: 0.6905
Epoch 123/200
80/80 [==============================] - 12s 150ms/step - loss: 0.2025 - acc
uracy: 0.9348 - val_loss: 1.2882 - val_accuracy: 0.6945
Epoch 124/200
80/80 [==============================] - 12s 147ms/step - loss: 0.2277 - acc
uracy: 0.9265 - val_loss: 1.3062 - val_accuracy: 0.6960
Epoch 125/200
80/80 [==============================] - 12s 144ms/step - loss: 0.2038 - acc
uracy: 0.9345 - val_loss: 1.3293 - val_accuracy: 0.6885
Epoch 126/200
80/80 [==============================] - 11s 132ms/step - loss: 0.1973 - acc
uracy: 0.9374 - val_loss: 1.3679 - val_accuracy: 0.6875
Epoch 127/200
80/80 [==============================] - 10s 124ms/step - loss: 0.2038 - acc
uracy: 0.9344 - val_loss: 1.3546 - val_accuracy: 0.6910
Epoch 128/200
80/80 [==============================] - 10s 124ms/step - loss: 0.2190 - acc
uracy: 0.9312 - val_loss: 1.3373 - val_accuracy: 0.6920
```

```
Epoch 129/200
80/80 [==============================] - 12s 147ms/step - loss: 0.2108 - acc
uracy: 0.9360 - val_loss: 1.3334 - val_accuracy: 0.6945
Epoch 130/200
80/80 [==============================] - 10s 125ms/step - loss: 0.2063 - acc
uracy: 0.9348 - val_loss: 1.4117 - val_accuracy: 0.6825
Epoch 131/200
80/80 [==============================] - 11s 133ms/step - loss: 0.2107 - acc
uracy: 0.9315 - val_loss: 1.3914 - val_accuracy: 0.6855
Epoch 132/200
80/80 [==============================] - 10s 120ms/step - loss: 0.2029 - acc
uracy: 0.9389 - val_loss: 1.3427 - val_accuracy: 0.7075
Epoch 133/200
80/80 [==============================] - 10s 123ms/step - loss: 0.2113 - acc
uracy: 0.9345 - val_loss: 1.4772 - val_accuracy: 0.6775
Epoch 134/200
80/80 [==============================] - 10s 128ms/step - loss: 0.2221 - acc
uracy: 0.9317 - val_loss: 1.3301 - val_accuracy: 0.7045
Epoch 135/200
80/80 [==============================] - 11s 133ms/step - loss: 0.2144 - acc
uracy: 0.9369 - val_loss: 1.4606 - val_accuracy: 0.6785
Epoch 136/200
80/80 [==============================] - 9s 117ms/step - loss: 0.2281 - accu
racy: 0.9304 - val_loss: 1.3990 - val_accuracy: 0.6815
Epoch 137/200
80/80 [==============================] - 10s 121ms/step - loss: 0.1928 - acc
uracy: 0.9360 - val_loss: 1.3746 - val_accuracy: 0.6860
Epoch 138/200
80/80 [==============================] - 10s 126ms/step - loss: 0.1845 - acc
uracy: 0.9440 - val_loss: 1.4302 - val_accuracy: 0.6765
Epoch 139/200
80/80 [==============================] - 10s 123ms/step - loss: 0.1886 - acc
uracy: 0.9430 - val_loss: 1.4390 - val_accuracy: 0.6720
Epoch 140/200
80/80 [==============================] - 10s 130ms/step - loss: 0.1941 - acc
uracy: 0.9388 - val_loss: 1.3176 - val_accuracy: 0.6910
Epoch 141/200
80/80 [==============================] - 10s 125ms/step - loss: 0.1931 - acc
uracy: 0.9402 - val_loss: 1.3379 - val_accuracy: 0.6935
Epoch 142/200
80/80 [==============================] - 9s 114ms/step - loss: 0.1863 - accu
racy: 0.9409 - val_loss: 1.3390 - val_accuracy: 0.6950
Epoch 143/200
80/80 [==============================] - 9s 115ms/step - loss: 0.1817 - accu
racy: 0.9434 - val_loss: 1.3482 - val_accuracy: 0.6970
Epoch 144/200
80/80 [==============================] - 9s 114ms/step - loss: 0.1733 - accu
racy: 0.9450 - val_loss: 1.4115 - val_accuracy: 0.6865
Epoch 145/200
80/80 [==============================] - 10s 121ms/step - loss: 0.1851 - acc
uracy: 0.9438 - val_loss: 1.4274 - val_accuracy: 0.6925
Epoch 146/200
80/80 [==============================] - 9s 114ms/step - loss: 0.1798 - accu
racy: 0.9456 - val_loss: 1.4398 - val_accuracy: 0.6860
Epoch 147/200
80/80 [==============================] - 10s 119ms/step - loss: 0.1688 - acc
uracy: 0.9464 - val_loss: 1.3307 - val_accuracy: 0.7030
Epoch 148/200
80/80 [==============================] - 9s 113ms/step - loss: 0.1688 - accu
racy: 0.9475 - val_loss: 1.3717 - val_accuracy: 0.6860
Epoch 149/200
80/80 [==============================] - 9s 116ms/step - loss: 0.1863 - accu
racy: 0.9419 - val_loss: 1.3934 - val_accuracy: 0.7035
Epoch 150/200
```

```
80/80 [==============================] - 9s 111ms/step - loss: 0.1838 - accu
racy: 0.9425 - val_loss: 1.3785 - val_accuracy: 0.6895
Epoch 151/200
80/80 [==============================] - 9s 115ms/step - loss: 0.1755 - accu
racy: 0.9434 - val_loss: 1.4111 - val_accuracy: 0.6995
Epoch 152/200
80/80 [==============================] - 9s 113ms/step - loss: 0.1757 - accu
racy: 0.9455 - val_loss: 1.4267 - val_accuracy: 0.6875
Epoch 153/200
80/80 [==============================] - 10s 120ms/step - loss: 0.1593 - acc
uracy: 0.9503 - val_loss: 1.4792 - val_accuracy: 0.6835
Epoch 154/200
80/80 [==============================] - 9s 114ms/step - loss: 0.1617 - accu
racy: 0.9504 - val_loss: 1.5238 - val_accuracy: 0.6760
Epoch 155/200
80/80 [==============================] - 9s 114ms/step - loss: 0.1819 - accu
racy: 0.9420 - val_loss: 1.3989 - val_accuracy: 0.6990
Epoch 156/200
80/80 [==============================] - 9s 112ms/step - loss: 0.1774 - accu
racy: 0.9435 - val_loss: 1.3841 - val_accuracy: 0.6930
Epoch 157/200
80/80 [==============================] - 9s 115ms/step - loss: 0.1780 - accu
racy: 0.9454 - val_loss: 1.4139 - val_accuracy: 0.6820
Epoch 158/200
80/80 [==============================] - 9s 113ms/step - loss: 0.1761 - accu
racy: 0.9459 - val_loss: 1.3682 - val_accuracy: 0.6975
Epoch 159/200
80/80 [==============================] - 9s 114ms/step - loss: 0.1707 - accu
racy: 0.9469 - val_loss: 1.5332 - val_accuracy: 0.6880
Epoch 160/200
80/80 [==============================] - 9s 112ms/step - loss: 0.1728 - accu
racy: 0.9450 - val_loss: 1.4395 - val_accuracy: 0.6830
Epoch 161/200
80/80 [==============================] - 9s 116ms/step - loss: 0.1814 - accu
racy: 0.9433 - val_loss: 1.4639 - val_accuracy: 0.6840
Epoch 162/200
80/80 [==============================] - 9s 114ms/step - loss: 0.1631 - accu
racy: 0.9515 - val_loss: 1.4471 - val_accuracy: 0.6890
Epoch 163/200
80/80 [==============================] - 9s 117ms/step - loss: 0.1664 - accu
racy: 0.9474 - val_loss: 1.5533 - val_accuracy: 0.6615
Epoch 164/200
80/80 [==============================] - 9s 111ms/step - loss: 0.1678 - accu
racy: 0.9485 - val_loss: 1.4623 - val_accuracy: 0.6810
Epoch 165/200
80/80 [==============================] - 9s 114ms/step - loss: 0.1574 - accu
racy: 0.9516 - val_loss: 1.4819 - val_accuracy: 0.6845
Epoch 166/200
80/80 [==============================] - 9s 113ms/step - loss: 0.1540 - accu
racy: 0.9505 - val_loss: 1.4361 - val_accuracy: 0.7015
Epoch 167/200
80/80 [==============================] - 9s 115ms/step - loss: 0.1729 - accu
racy: 0.9444 - val_loss: 1.4481 - val_accuracy: 0.6850
Epoch 168/200
80/80 [==============================] - 9s 111ms/step - loss: 0.1586 - accu
racy: 0.9530 - val_loss: 1.4276 - val_accuracy: 0.6855
Epoch 169/200
80/80 [==============================] - 9s 115ms/step - loss: 0.1667 - accu
racy: 0.9500 - val_loss: 1.4334 - val_accuracy: 0.6860
Epoch 170/200
80/80 [==============================] - 9s 112ms/step - loss: 0.1507 - accu
racy: 0.9536 - val_loss: 1.4491 - val_accuracy: 0.6910
Epoch 171/200
80/80 [==============================] - 9s 113ms/step - loss: 0.1492 - accu
```

```
racy: 0.9506 - val_loss: 1.3585 - val_accuracy: 0.7030
Epoch 172/200
80/80 [==============================] - 9s 113ms/step - loss: 0.1341 - accu
racy: 0.9579 - val_loss: 1.3904 - val_accuracy: 0.7035
Epoch 173/200
80/80 [==============================] - 9s 116ms/step - loss: 0.1303 - accu
racy: 0.9591 - val_loss: 1.4646 - val_accuracy: 0.6840
Epoch 174/200
80/80 [==============================] - 9s 111ms/step - loss: 0.1343 - accu
racy: 0.9591 - val_loss: 1.4806 - val_accuracy: 0.6850
Epoch 175/200
80/80 [==============================] - 9s 114ms/step - loss: 0.1452 - accu
racy: 0.9551 - val_loss: 1.5111 - val_accuracy: 0.6810
Epoch 176/200
80/80 [==============================] - 9s 111ms/step - loss: 0.1408 - accu
racy: 0.9575 - val_loss: 1.4860 - val_accuracy: 0.6985
Epoch 177/200
80/80 [==============================] - 9s 111ms/step - loss: 0.1422 - accu
racy: 0.9538 - val_loss: 1.4659 - val_accuracy: 0.6945
Epoch 178/200
80/80 [==============================] - 9s 111ms/step - loss: 0.1414 - accu
racy: 0.9559 - val_loss: 1.4358 - val_accuracy: 0.6975
Epoch 179/200
80/80 [==============================] - 9s 114ms/step - loss: 0.1459 - accu
racy: 0.9517 - val_loss: 1.6053 - val_accuracy: 0.6745
Epoch 180/200
80/80 [==============================] - 9s 111ms/step - loss: 0.1428 - accu
racy: 0.9549 - val_loss: 1.4620 - val_accuracy: 0.6865
Epoch 181/200
80/80 [==============================] - 9s 115ms/step - loss: 0.1369 - accu
racy: 0.9584 - val_loss: 1.4135 - val_accuracy: 0.7110
Epoch 182/200
80/80 [==============================] - 9s 111ms/step - loss: 0.1281 - accu
racy: 0.9630 - val_loss: 1.4867 - val_accuracy: 0.6880
Epoch 183/200
80/80 [==============================] - 9s 114ms/step - loss: 0.1397 - accu
racy: 0.9557 - val_loss: 1.4706 - val_accuracy: 0.6980
Epoch 184/200
80/80 [==============================] - 9s 112ms/step - loss: 0.1407 - accu
racy: 0.9572 - val_loss: 1.5383 - val_accuracy: 0.6820
Epoch 185/200
80/80 [==============================] - 9s 115ms/step - loss: 0.1413 - accu
racy: 0.9576 - val_loss: 1.6651 - val_accuracy: 0.6635
Epoch 186/200
80/80 [==============================] - 9s 112ms/step - loss: 0.1529 - accu
racy: 0.9534 - val_loss: 1.5077 - val_accuracy: 0.6925
Epoch 187/200
80/80 [==============================] - 9s 112ms/step - loss: 0.1272 - accu
racy: 0.9589 - val_loss: 1.5180 - val_accuracy: 0.6850
Epoch 188/200
80/80 [==============================] - 9s 112ms/step - loss: 0.1432 - accu
racy: 0.9578 - val_loss: 1.4606 - val_accuracy: 0.6945
Epoch 189/200
80/80 [==============================] - 9s 114ms/step - loss: 0.1413 - accu
racy: 0.9582 - val_loss: 1.4232 - val_accuracy: 0.7010
Epoch 190/200
80/80 [==============================] - 9s 109ms/step - loss: 0.1439 - accu
racy: 0.9530 - val_loss: 1.4953 - val_accuracy: 0.6880
Epoch 191/200
80/80 [==============================] - 9s 115ms/step - loss: 0.1423 - accu
racy: 0.9548 - val_loss: 1.4508 - val_accuracy: 0.7005
Epoch 192/200
80/80 [==============================] - 9s 112ms/step - loss: 0.1358 - accu
racy: 0.9599 - val_loss: 1.5702 - val_accuracy: 0.6795
```

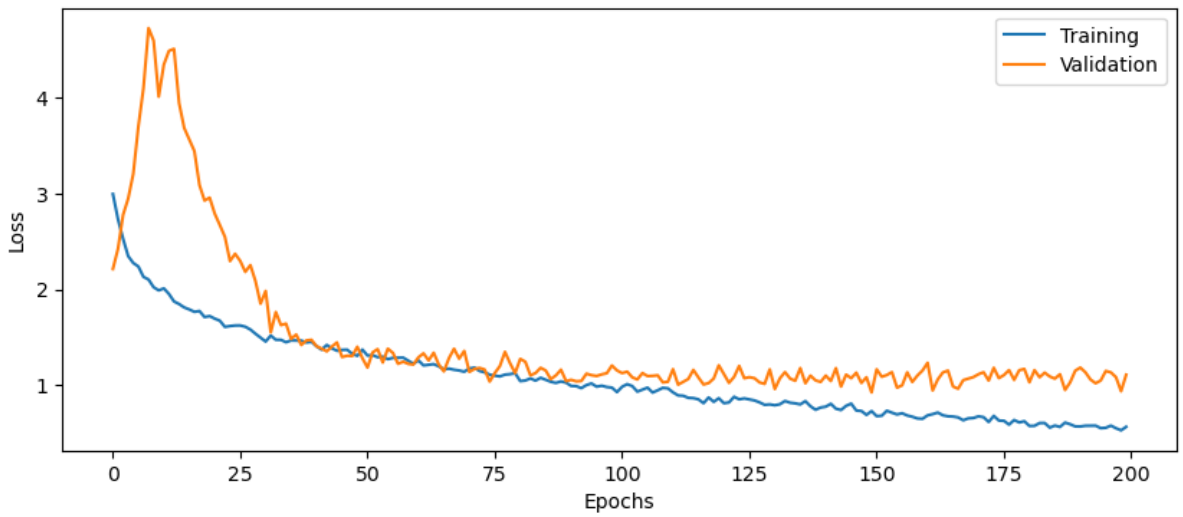
```
Epoch 193/200
80/80 [==============================] - 9s 115ms/step - loss: 0.1209 - accu
racy: 0.9625 - val_loss: 1.4845 - val_accuracy: 0.6945
Epoch 194/200
80/80 [==============================] - 9s 110ms/step - loss: 0.1402 - accu
racy: 0.9549 - val_loss: 1.6923 - val_accuracy: 0.6690
Epoch 195/200
80/80 [==============================] - 9s 112ms/step - loss: 0.1419 - accu
racy: 0.9570 - val_loss: 1.5415 - val_accuracy: 0.6890
Epoch 196/200
80/80 [==============================] - 9s 114ms/step - loss: 0.1414 - accu
racy: 0.9567 - val_loss: 1.5045 - val_accuracy: 0.6995
Epoch 197/200
80/80 [==============================] - 9s 116ms/step - loss: 0.1303 - accu
racy: 0.9617 - val_loss: 1.5207 - val_accuracy: 0.6975
Epoch 198/200
80/80 [==============================] - 10s 119ms/step - loss: 0.1285 - acc
uracy: 0.9590 - val_loss: 1.4737 - val_accuracy: 0.6945
Epoch 199/200
80/80 [==============================] - 9s 118ms/step - loss: 0.1203 - accu
racy: 0.9617 - val_loss: 1.4651 - val_accuracy: 0.6900
Epoch 200/200
80/80 [==============================] - 9s 114ms/step - loss: 0.1199 - accu
racy: 0.9635 - val_loss: 1.5540 - val_accuracy: 0.6950
```

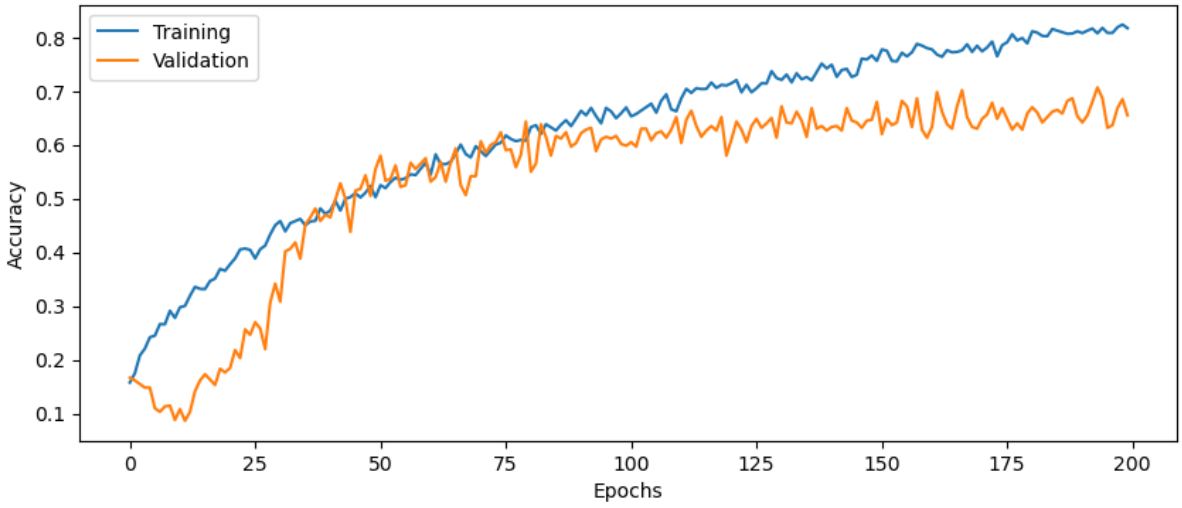
In [ ]:

In [36]:
```
# Check if there is still a big difference in accuracy for original and rota

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

```
Test loss: 1.1374
Test accuracy: 0.6670
Test loss: 2.4780
Test accuracy: 0.3480
```

In [37]:
```
# 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 [38]:
```
# 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 20ms/step
```

In [39]:
```
# 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[
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?

Question 26: Is it possible to design a CNN that can be trained on images of one size, and then applied to an image of any size? How?

Answer

- A25 : This CNN cannot be applied to images of another size because of the Dense layers present in the model. These layers require a particular shape of input.
- A26 : Yes, a full convolutional CNN would be able to predict on an image of any size because the conv2D layers are independent of the input shape.

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?

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

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

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

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

Anwer

- A27 : 53
- A28 : 25,583,592 Trainable Parameters
- A29 : Input shape expected is (224, 224, 3)
- A30 : Because computing and storing the 2nd derivatives for over 25 million parameters is expensive computationally and requires a lot of memory.

In [78]:
```
# 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
from tensorflow.keras.preprocessing import image
from tensorflow.keras.applications.resnet50 import ResNet50, preprocess_inpu
import numpy as np
import os

pretrain_model = ResNet50(weights='imagenet')
current_dir = os.getcwd()
dir_path = os.fsencode(f"{current_dir}/pictures")
image_list = []
files = os.listdir(dir_path)
for i in range(0,5):
    picture = image.load_img(f"{current_dir}/pictures/{files[i].decode('utf-
    image_list.append(picture)
    picture = image.img_to_array(picture)
    picture = np.expand_dims(picture, axis=0)
    picture = preprocess_input(picture)

    predicted_labels = pretrain_model.predict(picture)
    print(files[i].decode('utf-8'))
    print('Predicted:', decode_predictions(predicted_labels))
```

```
1/1 [==============================] - 1s 655ms/step
birdplane.jpg
Predicted: [[('n01608432', 'kite', 0.66807944), ('n01614925', 'bald_eagle',
0.18754959), ('n04266014', 'space_shuttle', 0.117994346), ('n04552348', 'war
plane', 0.012906263), ('n02058221', 'albatross', 0.00258241)]]
1/1 [==============================] - 0s 62ms/step
flyingcar.jpg
Predicted: [[('n04552348', 'warplane', 0.819623), ('n04592741', 'wing', 0.08
372145), ('n02690373', 'airliner', 0.07428714), ('n04347754', 'submarine',
0.007198979), ('n03967562', 'plow', 0.0066802725)]]
1/1 [==============================] - 0s 66ms/step
framework_laptop.jpg
Predicted: [[('n03832673', 'notebook', 0.5465459), ('n03642806', 'laptop',
0.3740683), ('n03485407', 'hand-held_computer', 0.009557434), ('n02966687',
"carpenter's_kit", 0.008635032), ('n03492542', 'hard_disc', 0.006197259)]]
1/1 [==============================] - 0s 66ms/step
ltt waterbottle.jpg
Predicted: [[('n02815834', 'beaker', 0.48271772), ('n04557648', 'water_bottl
e', 0.28567883), ('n03063689', 'coffeepot', 0.02760419), ('n03063599', 'coff
ee_mug', 0.026299164), ('n03843555', 'oil_filter', 0.021989664)]]
1/1 [==============================] - 0s 62ms/step
unidog.jpg
Predicted: [[('n02100735', 'English_setter', 0.61889803), ('n02091831', 'Sal
uki', 0.05842548), ('n02086910', 'papillon', 0.049064197), ('n02091244', 'Ib
izan_hound', 0.0281835), ('n02085620', 'Chihuahua', 0.021989055)]]
```