CNN_Lab_2024

April 23, 2024

1 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.

1.1 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).

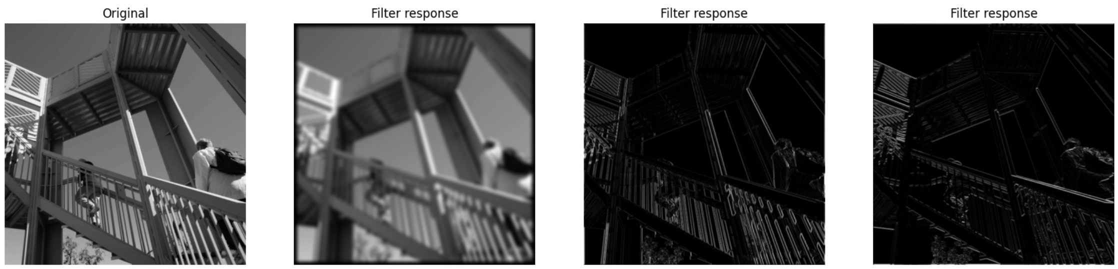
[1]:
```python
# This cell is finished
from scipy import signal
import numpy as np

# Get a test image
from scipy import misc
image = misc.ascent()

# Define a help function for creating a Gaussian filter
def matlab_style_gauss2D(shape=(3,3),sigma=0.5):
    """
    2D gaussian mask - should give the same result as MATLAB's
    fspecial('gaussian',[shape],[sigma])
    """
    m,n = [(ss-1.)/2. for ss in shape]
    y,x = np.ogrid[-m:m+1,-n:n+1]
```

```
    h = np.exp( -(x*x + y*y) / (2.*sigma*sigma) )
    h[ h < np.finfo(h.dtype).eps*h.max() ] = 0
    sumh = h.sum()
    if sumh != 0:
        h /= sumh
    return h

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

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

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

```
/tmp/ipykernel_912095/3554125653.py:7: DeprecationWarning: 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()
```

[2]:
```
# 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)
```

[3]:
```
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()
```



1.2 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?

[4]:
```python
# Your code for checking sizes of image and filter responses

print(f"Shape of the orginal image: {image.shape}")
print(f"Size of the orginal image (total pixels): {image.size}")
print(f"Dimensions of data: {image.ndim}, 2d-data is only one channel.")
print("***************************************************")
print(f"Size of guassFilter: {gaussFilter.size}")
print(f"Size of SobelX: {sobelX.size}")
print(f"Size of SobelY: {sobelY.size}")
print("***************************************************")
```

```
Shape of the orginal image: (512, 512)
Size of the orginal image (total pixels): 262144
Dimensions of data: 2, 2d-data is only one channel.
***************************************************
Size of guassFilter: 225
Size of SobelX: 9
Size of SobelY: 9
***************************************************
```

Answer: Question 1

- All filters goes through all pixels and calculates the scalar product between filter coefficients and signal values. The guassian filter has larger values in the center of the filter and smaller values at the edges, which result in a blurry image. The SobelX filter detects vertical lines and the SobelY filter detects horizontal lines.

Question 2

- The shape of the orginal image is 512 x 512 (262 144 total pixels). The image have one channel and a color image normally have three channels.

Question 3

- The size of guassFilter is 15 x 15 (225 pixels), the size of SobelX is 3 x 3 (9 pixels) and the size of SobelY is 3 x 3 (9 pixels).

Question 4

- The size of the filter response if mode = 'same' is the same size as orginal image (512 x 512).

Question 5

- The size of the filter response if mode = 'valid' is 510 x 510, since we can not compute filter values in the edges of the orignal image.

Question 6

- It can be problem since each layer will reduce the image size so in a CNN with many layers the last layer can have a lot smaller images than the orginal image.

1.3 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.

[50]:
```
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 being used
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[50], line 17
     15 # Allow growth of GPU memory, otherwise it will always look like all the␣
 ↪memory 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
```

1.4 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: Question 7

- The fliter for a color image have size 7 x 7 x 3, since it have three different channels.

Question 8

- The operation performerd by Conv2D is 2D convolution over the image. Conv2D does the same operations with different padding as standard ("valid"). Conv2d learns the filter, while signal.convolve2d takes a predefined filter as input.

Question 9

- We think that in a CPU 1000 images is slower than 3 images, since it can not convolve all images at once. In GPU it dependends how many cores there is, if it is over 1000 cores then it can run 1000 image as fast as 3 images, therefore the GPU should be fatser than CPU.

1.5 Part 5: Load data

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

[5]:
```python
from keras.datasets import cifar10
import numpy as np

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

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

```python
print("Training images have size {} and labels have size {} ".format(Xtrain.
 ↪shape, Ytrain.shape))
print("Test images have size {} and labels have size {} \n ".format(Xtest.
 ↪shape, Ytest.shape))

# 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, Ytrain.shape))
print("Reduced test images have size %s and labels have size %s \n" % (Xtest.
 ↪shape, Ytest.shape))

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

```
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
```

1.6 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.

[6]:
```
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()
```

1.7 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

[7]:
```
from sklearn.model_selection import train_test_split

# Your code for splitting the dataset
# Labb assistant told 25 % of the data = 25 % of the traindata.

Xtrain, Xval, Ytrain, Yval = train_test_split(Xtrain, Ytrain, test_size=0.25␣
 ↪,random_state=13)
```

```
# Print the size of training data, validation data and test data
print("Training images have size {}".format(Xtrain.shape))
print("Test images have size {}".format(Xtest.shape))
print("Validation images have size {}\n ".format(Xval.shape))
```

```
Training images have size (7500, 32, 32, 3)
Test images have size (2000, 32, 32, 3)
Validation images have size (2500, 32, 32, 3)
```

1.8 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.

[8]:
```
# 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
```

1.9 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

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

# Print shapes before converting the labels
print("Train labels have size {} ".format(Ytrain.shape))
print("Test labels have size {} ".format(Ytest.shape))
print("Validation labels have size {} \n ".format(Yval.shape))


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


# Print shapes after converting the labels
```

```
print("Train labels have size {} ".format(Ytrain.shape))
print("Test labels have size {} ".format(Ytest.shape))
print("Validation labels have size {} \n ".format(Yval.shape))
```

```
Train labels have size (7500, 1)
Test labels have size (2000, 1)
Validation labels have size (2500, 1)

Train labels have size (7500, 10)
Test labels have size (2000, 10)
Validation labels have size (2500, 10)
```

1.10 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

[10]:
```
from keras.models import Sequential, Model
from keras.layers import Input, Conv2D, BatchNormalization, MaxPooling2D,␣
↪Flatten, Dense, Dropout
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=50, use_dropout=False, learning_rate=0.01):

    # Setup a sequential model
    model = Sequential()

    # Add first convolutional layer to the model, requires input shape
    model.add(Conv2D(filters = n_filters, kernel_size = (3,3),
                     padding = "same",
                     activation = "relu",
                     input_shape = input_shape))

    model.add(BatchNormalization())
    model.add(MaxPooling2D())



    # Add remaining convolutional layers to the model, the number of filters␣
↪should increase a factor 2 for each layer
    for i in range(n_conv_layers-1):

        model.add(Conv2D(filters = n_filters * (1+i), kernel_size = (3,3),
                     padding = "same",
                     activation = "relu"))

        model.add(BatchNormalization())
        model.add(MaxPooling2D())




    # 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):
            model.add(Dropout(0.5))



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

    # Compile model
    model.compile(loss=CategoricalCrossentropy(), optimizer =␣
 ↪Adam(learning_rate = learning_rate), metrics = ["accuracy"])

    return model
```

[11]:

```
# 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()
```

1.11 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

1.12 2 convolutional layers, no intermediate dense layers

[12]:
```
# Setup some training parameters
batch_size = 100
epochs = 20
input_shape = (32,32,3)

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

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

```
Epoch 1/20
75/75 [==============================] - 4s 40ms/step - loss: 2.2179 - accuracy:
0.3349 - val_loss: 1.7909 - val_accuracy: 0.3372
Epoch 2/20
75/75 [==============================] - 3s 36ms/step - loss: 1.5009 - accuracy:
0.4741 - val_loss: 1.5544 - val_accuracy: 0.4444
Epoch 3/20
75/75 [==============================] - 3s 36ms/step - loss: 1.2826 - accuracy:
0.5451 - val_loss: 1.5482 - val_accuracy: 0.4336
Epoch 4/20
75/75 [==============================] - 3s 36ms/step - loss: 1.1915 - accuracy:
0.5761 - val_loss: 1.4539 - val_accuracy: 0.4792
Epoch 5/20
75/75 [==============================] - 3s 41ms/step - loss: 1.1062 - accuracy:
0.6075 - val_loss: 1.3449 - val_accuracy: 0.5144
Epoch 6/20
75/75 [==============================] - 3s 36ms/step - loss: 1.0149 - accuracy:
0.6435 - val_loss: 1.2941 - val_accuracy: 0.5408
Epoch 7/20
75/75 [==============================] - 3s 36ms/step - loss: 0.9780 - accuracy:
```

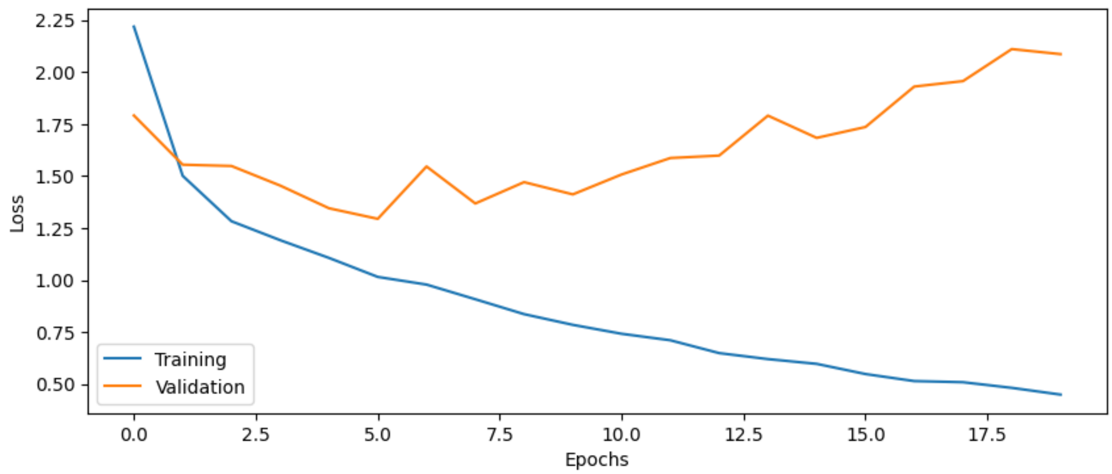
```
0.6519 - val_loss: 1.5458 - val_accuracy: 0.4984
Epoch 8/20
75/75 [==============================] - 3s 37ms/step - loss: 0.9075 - accuracy:
0.6831 - val_loss: 1.3679 - val_accuracy: 0.5412
Epoch 9/20
75/75 [==============================] - 3s 37ms/step - loss: 0.8359 - accuracy:
0.7043 - val_loss: 1.4705 - val_accuracy: 0.5448
Epoch 10/20
75/75 [==============================] - 3s 36ms/step - loss: 0.7845 - accuracy:
0.7236 - val_loss: 1.4116 - val_accuracy: 0.5668
Epoch 11/20
75/75 [==============================] - 3s 34ms/step - loss: 0.7415 - accuracy:
0.7445 - val_loss: 1.5073 - val_accuracy: 0.5396
Epoch 12/20
75/75 [==============================] - 4s 55ms/step - loss: 0.7103 - accuracy:
0.7432 - val_loss: 1.5865 - val_accuracy: 0.5512
Epoch 13/20
75/75 [==============================] - 2s 33ms/step - loss: 0.6487 - accuracy:
0.7729 - val_loss: 1.5982 - val_accuracy: 0.5532
Epoch 14/20
75/75 [==============================] - 3s 35ms/step - loss: 0.6198 - accuracy:
0.7804 - val_loss: 1.7902 - val_accuracy: 0.5372
Epoch 15/20
75/75 [==============================] - 2s 33ms/step - loss: 0.5969 - accuracy:
0.7940 - val_loss: 1.6833 - val_accuracy: 0.5632
Epoch 16/20
75/75 [==============================] - 3s 36ms/step - loss: 0.5480 - accuracy:
0.8096 - val_loss: 1.7355 - val_accuracy: 0.5692
Epoch 17/20
75/75 [==============================] - 3s 35ms/step - loss: 0.5142 - accuracy:
0.8133 - val_loss: 1.9300 - val_accuracy: 0.5560
Epoch 18/20
75/75 [==============================] - 3s 35ms/step - loss: 0.5087 - accuracy:
0.8161 - val_loss: 1.9562 - val_accuracy: 0.5484
Epoch 19/20
75/75 [==============================] - 3s 34ms/step - loss: 0.4816 - accuracy:
0.8220 - val_loss: 2.1101 - val_accuracy: 0.5352
Epoch 20/20
75/75 [==============================] - 3s 35ms/step - loss: 0.4492 - accuracy:
0.8372 - val_loss: 2.0859 - val_accuracy: 0.5628
```

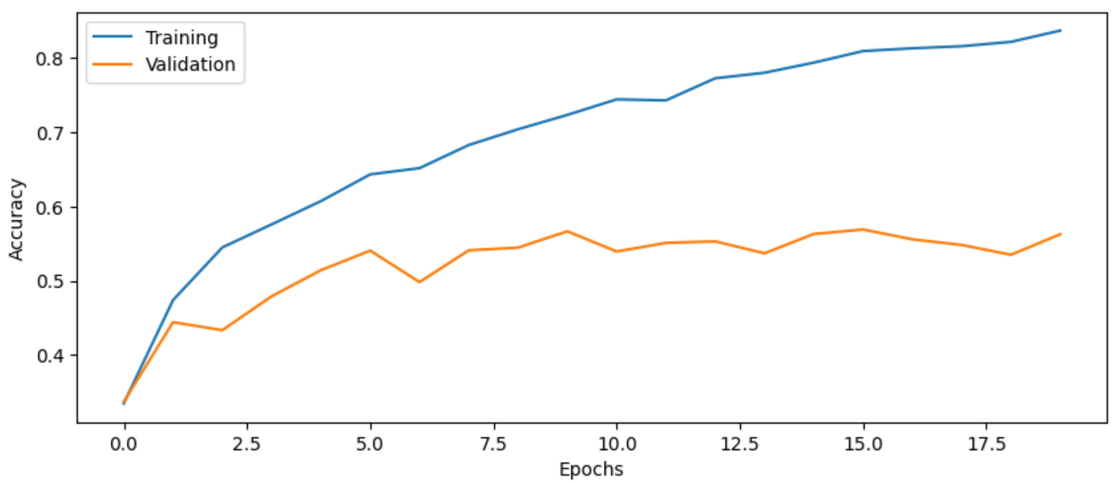
[13]:
```python
# Evaluate the trained model on test set, not used in training or validation
score = model1.evaluate(Xtest, Ytest)
print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])
```

```
63/63 [==============================] - 0s 4ms/step - loss: 2.1313 - accuracy:
0.5505
```

```
Test loss: 2.1313
Test accuracy: 0.5505
```

[14]:
```python
# Plot the history from the training run
plot_results(history1)
```





1.13 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?

Answer: Test accuracy is around 0.5505, it is better than random guessing, but we are not satisfied.

Question 10

- The training accuracy is 0.8372 and the difference is around 0.29.

Question 11

- We use a smaller batch size in this laboration, since we had 92 features per observation and in this lab we have 32 x 32 x 3 features which alot more than 92. This means our data is larger and therefore we need to use smaller batches to fit everything in memory.

1.14 2 convolutional layers, 1 intermediate dense layer (50 nodes)

[15]:
```
# Setup some training parameters
batch_size = 100
epochs = 20
input_shape = (32,32,3)

# Build model
model2 = build_CNN(input_shape = 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, validation_data = (Xval, Yval),␣
↪batch_size = batch_size, epochs = epochs)
```

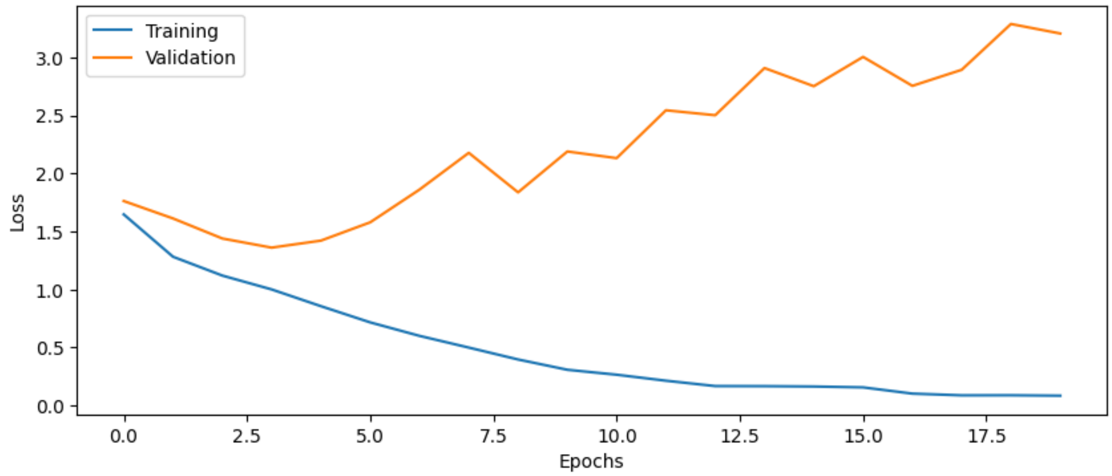
```
Epoch 1/20
75/75 [==============================] - 4s 36ms/step - loss: 1.6468 - accuracy:
0.4121 - val_loss: 1.7621 - val_accuracy: 0.3556
Epoch 2/20
75/75 [==============================] - 3s 36ms/step - loss: 1.2821 - accuracy:
0.5376 - val_loss: 1.6120 - val_accuracy: 0.4160
Epoch 3/20
75/75 [==============================] - 3s 37ms/step - loss: 1.1200 - accuracy:
0.5973 - val_loss: 1.4394 - val_accuracy: 0.4892
Epoch 4/20
75/75 [==============================] - 3s 38ms/step - loss: 1.0006 - accuracy:
0.6417 - val_loss: 1.3607 - val_accuracy: 0.5136
Epoch 5/20
75/75 [==============================] - 3s 36ms/step - loss: 0.8563 - accuracy:
0.6976 - val_loss: 1.4210 - val_accuracy: 0.5368
Epoch 6/20
75/75 [==============================] - 3s 36ms/step - loss: 0.7167 - accuracy:
0.7504 - val_loss: 1.5787 - val_accuracy: 0.5244
Epoch 7/20
75/75 [==============================] - 3s 35ms/step - loss: 0.6006 - accuracy:
0.7836 - val_loss: 1.8607 - val_accuracy: 0.5096
Epoch 8/20
```

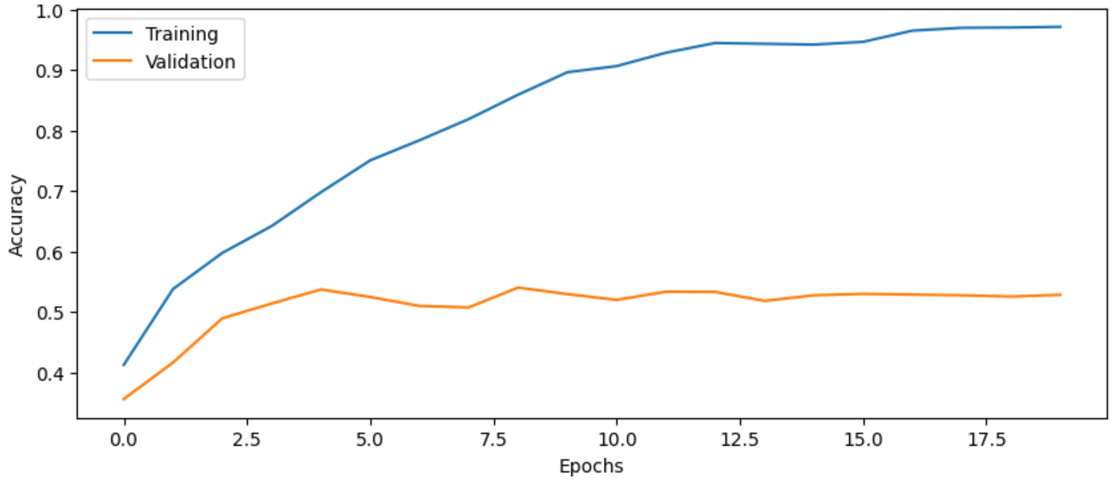
```
75/75 [==============================] - 3s 36ms/step - loss: 0.4996 - accuracy:
0.8187 - val_loss: 2.1775 - val_accuracy: 0.5072
Epoch 9/20
75/75 [==============================] - 3s 36ms/step - loss: 0.3971 - accuracy:
0.8589 - val_loss: 1.8368 - val_accuracy: 0.5400
Epoch 10/20
75/75 [==============================] - 3s 35ms/step - loss: 0.3084 - accuracy:
0.8961 - val_loss: 2.1890 - val_accuracy: 0.5292
Epoch 11/20
75/75 [==============================] - 3s 36ms/step - loss: 0.2658 - accuracy:
0.9063 - val_loss: 2.1317 - val_accuracy: 0.5196
Epoch 12/20
75/75 [==============================] - 3s 35ms/step - loss: 0.2142 - accuracy:
0.9284 - val_loss: 2.5440 - val_accuracy: 0.5332
Epoch 13/20
75/75 [==============================] - 3s 36ms/step - loss: 0.1680 - accuracy:
0.9445 - val_loss: 2.5014 - val_accuracy: 0.5328
Epoch 14/20
75/75 [==============================] - 3s 36ms/step - loss: 0.1672 - accuracy:
0.9432 - val_loss: 2.9079 - val_accuracy: 0.5180
Epoch 15/20
75/75 [==============================] - 3s 36ms/step - loss: 0.1642 - accuracy:
0.9419 - val_loss: 2.7515 - val_accuracy: 0.5272
Epoch 16/20
75/75 [==============================] - 3s 36ms/step - loss: 0.1567 - accuracy:
0.9464 - val_loss: 3.0038 - val_accuracy: 0.5296
Epoch 17/20
75/75 [==============================] - 3s 35ms/step - loss: 0.1030 - accuracy:
0.9649 - val_loss: 2.7545 - val_accuracy: 0.5284
Epoch 18/20
75/75 [==============================] - 3s 41ms/step - loss: 0.0882 - accuracy:
0.9696 - val_loss: 2.8927 - val_accuracy: 0.5272
Epoch 19/20
75/75 [==============================] - 3s 39ms/step - loss: 0.0887 - accuracy:
0.9701 - val_loss: 3.2875 - val_accuracy: 0.5252
Epoch 20/20
75/75 [==============================] - 3s 37ms/step - loss: 0.0852 - accuracy:
0.9712 - val_loss: 3.2056 - val_accuracy: 0.5280
```

[16]:
```python
score = model2.evaluate(Xtest, Ytest)
print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])
```

```
63/63 [==============================] - 0s 4ms/step - loss: 3.1830 - accuracy:
0.5325
Test loss: 3.1830
Test accuracy: 0.5325
```

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





1.15 4 convolutional layers, 1 intermediate dense layer (50 nodes)

```
[18]: # Setup some training parameters
batch_size = 100
epochs = 20
input_shape = (32,32,3)

# Build model
model3 = build_CNN(input_shape = 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, validation_data = (Xval, Yval),␣
↪batch_size = batch_size, epochs = epochs)
```

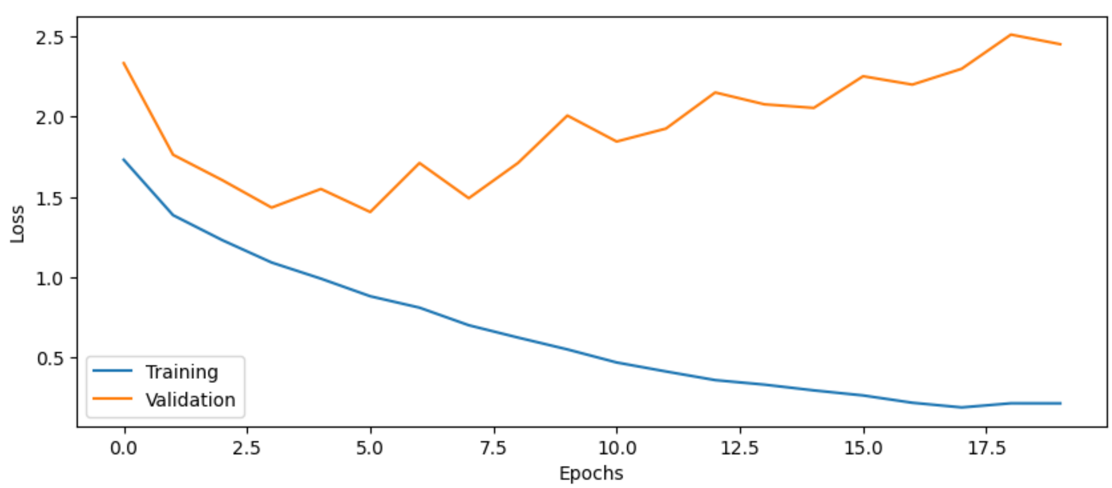
```
Epoch 1/20
75/75 [==============================] - 5s 44ms/step - loss: 1.7323 - accuracy:
0.3639 - val_loss: 2.3340 - val_accuracy: 0.2360
Epoch 2/20
75/75 [==============================] - 3s 41ms/step - loss: 1.3876 - accuracy:
0.4936 - val_loss: 1.7640 - val_accuracy: 0.3904
Epoch 3/20
75/75 [==============================] - 3s 41ms/step - loss: 1.2318 - accuracy:
0.5577 - val_loss: 1.6055 - val_accuracy: 0.4552
Epoch 4/20
75/75 [==============================] - 3s 42ms/step - loss: 1.0926 - accuracy:
0.6083 - val_loss: 1.4347 - val_accuracy: 0.5268
Epoch 5/20
75/75 [==============================] - 3s 41ms/step - loss: 0.9922 - accuracy:
0.6432 - val_loss: 1.5507 - val_accuracy: 0.5036
Epoch 6/20
75/75 [==============================] - 3s 43ms/step - loss: 0.8826 - accuracy:
0.6913 - val_loss: 1.4072 - val_accuracy: 0.5592
Epoch 7/20
75/75 [==============================] - 3s 43ms/step - loss: 0.8115 - accuracy:
0.7189 - val_loss: 1.7123 - val_accuracy: 0.5356
Epoch 8/20
75/75 [==============================] - 3s 41ms/step - loss: 0.7020 - accuracy:
0.7499 - val_loss: 1.4931 - val_accuracy: 0.5624
Epoch 9/20
75/75 [==============================] - 3s 43ms/step - loss: 0.6251 - accuracy:
0.7800 - val_loss: 1.7130 - val_accuracy: 0.5412
Epoch 10/20
75/75 [==============================] - 3s 44ms/step - loss: 0.5515 - accuracy:
0.8055 - val_loss: 2.0080 - val_accuracy: 0.5376
Epoch 11/20
75/75 [==============================] - 3s 43ms/step - loss: 0.4701 - accuracy:
0.8360 - val_loss: 1.8461 - val_accuracy: 0.5676
Epoch 12/20
75/75 [==============================] - 3s 43ms/step - loss: 0.4141 - accuracy:
0.8516 - val_loss: 1.9267 - val_accuracy: 0.5624
Epoch 13/20
75/75 [==============================] - 3s 46ms/step - loss: 0.3603 - accuracy:
0.8724 - val_loss: 2.1517 - val_accuracy: 0.5600
Epoch 14/20
75/75 [==============================] - 3s 45ms/step - loss: 0.3321 - accuracy:
0.8828 - val_loss: 2.0778 - val_accuracy: 0.5704
```

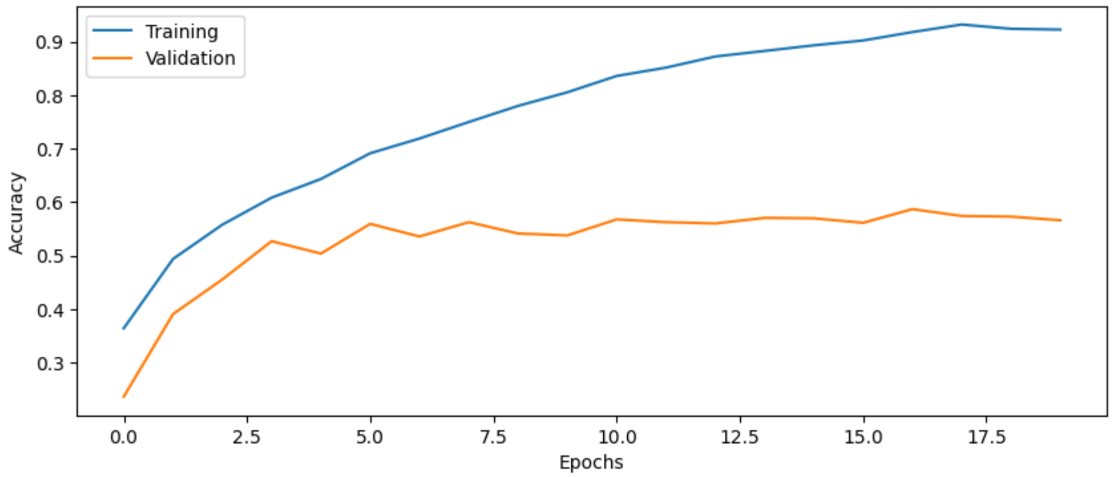
```
Epoch 15/20
75/75 [==============================] - 3s 42ms/step - loss: 0.2963 - accuracy:
0.8933 - val_loss: 2.0556 - val_accuracy: 0.5696
Epoch 16/20
75/75 [==============================] - 5s 61ms/step - loss: 0.2648 - accuracy:
0.9024 - val_loss: 2.2527 - val_accuracy: 0.5612
Epoch 17/20
75/75 [==============================] - 3s 43ms/step - loss: 0.2193 - accuracy:
0.9181 - val_loss: 2.2006 - val_accuracy: 0.5868
Epoch 18/20
75/75 [==============================] - 3s 42ms/step - loss: 0.1902 - accuracy:
0.9323 - val_loss: 2.2991 - val_accuracy: 0.5740
Epoch 19/20
75/75 [==============================] - 3s 45ms/step - loss: 0.2159 - accuracy:
0.9243 - val_loss: 2.5121 - val_accuracy: 0.5728
Epoch 20/20
75/75 [==============================] - 4s 52ms/step - loss: 0.2155 - accuracy:
0.9228 - val_loss: 2.4522 - val_accuracy: 0.5660
```

[19]:
```
# 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 5ms/step - loss: 2.6108 - accuracy:
0.5430
Test loss: 2.6108
Test accuracy: 0.5430
```

[20]:
```
# Plot the history from the training run
plot_results(history3)
```





1.16 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?

[21]:
```
# 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, 16)        2320

 batch_normalization_6 (Batc  (None, 16, 16, 16)       64
 hNormalization)

 max_pooling2d_5 (MaxPooling  (None, 8, 8, 16)         0
 2D)

 conv2d_6 (Conv2D)           (None, 8, 8, 32)          4640

 batch_normalization_7 (Batc  (None, 8, 8, 32)         128
 hNormalization)

 max_pooling2d_6 (MaxPooling  (None, 4, 4, 32)         0
 2D)

 conv2d_7 (Conv2D)           (None, 4, 4, 48)          13872

 batch_normalization_8 (Batc  (None, 4, 4, 48)         192
 hNormalization)

 max_pooling2d_7 (MaxPooling  (None, 2, 2, 48)         0
 2D)

 flatten_2 (Flatten)         (None, 192)               0

 dense_3 (Dense)             (None, 50)                9650

 batch_normalization_9 (Batc  (None, 50)               200
 hNormalization)

 dense_4 (Dense)             (None, 10)                510

=================================================================
Total params: 32,088
Trainable params: 31,764
Non-trainable params: 324
_________________________________________________________________
```

Answer: Question 12

- The number of trainable parameters of our network have 31 764 parameters. The last convolutional layer contains most of the parameters, since it has most filters.

Question 13

- The input and output of a Conv2D layer is the same, because padding = "same". The dimensions of the input and output are 32 x 32 x 3.

Question 14

- The batch size is always the first dimension of each 4D tensor.

Question 15

- The number of channels in the output layer is 128.

Question 16

- The number of parameters is "the number of filters" times "**the number of channels**" times "the number of filter coefficients" plus bias.

Question 17

- Maxpooling reduce the image size by downsampling the input.

1.17 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?

1.18 4 convolutional layers, 1 intermediate dense layer (50 nodes), dropout

[22]:
```
# Setup some training parameters
batch_size = 100
epochs = 20
input_shape = (32,32,3)

# Build model
model4 = build_CNN(input_shape = input_shape, n_conv_layers = 4, n_dense_layers␣
↪= 1, n_nodes=50, use_dropout = True)

# Train the model  using training data and validation data
history4 = model4.fit(Xtrain, Ytrain, validation_data = (Xval, Yval),␣
↪batch_size = batch_size, epochs = epochs)
```

```
Epoch 1/20
75/75 [==============================] - 5s 49ms/step - loss: 2.0912 - accuracy:
0.2756 - val_loss: 2.5512 - val_accuracy: 0.1516
Epoch 2/20
75/75 [==============================] - 4s 48ms/step - loss: 1.6664 - accuracy:
0.3833 - val_loss: 2.0061 - val_accuracy: 0.2676
Epoch 3/20
75/75 [==============================] - 3s 41ms/step - loss: 1.5206 - accuracy:
0.4412 - val_loss: 1.5391 - val_accuracy: 0.4288
```

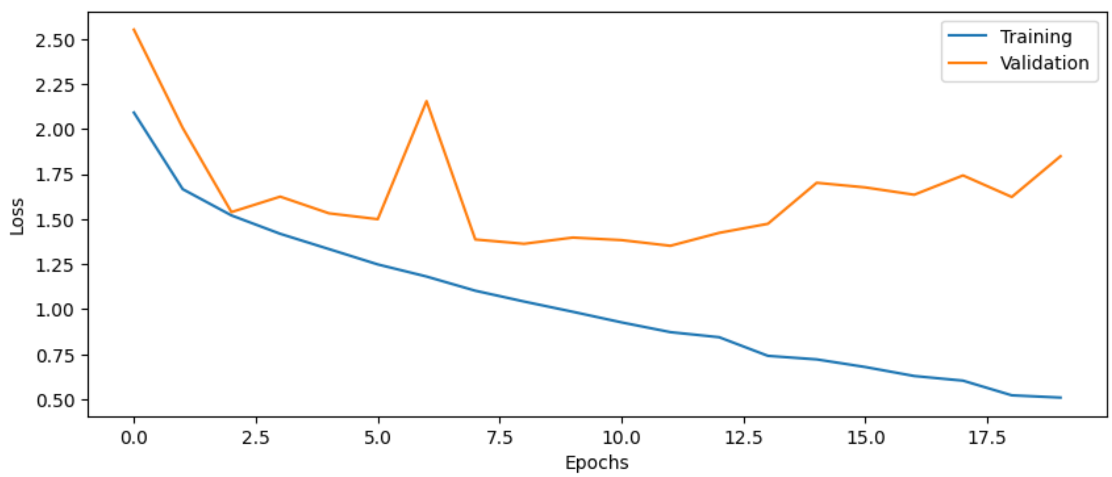
```
Epoch 4/20
75/75 [==============================] - 3s 42ms/step - loss: 1.4185 - accuracy:
0.4871 - val_loss: 1.6248 - val_accuracy: 0.4280
Epoch 5/20
75/75 [==============================] - 3s 42ms/step - loss: 1.3338 - accuracy:
0.5144 - val_loss: 1.5322 - val_accuracy: 0.4692
Epoch 6/20
75/75 [==============================] - 3s 44ms/step - loss: 1.2488 - accuracy:
0.5501 - val_loss: 1.4997 - val_accuracy: 0.5132
Epoch 7/20
75/75 [==============================] - 3s 44ms/step - loss: 1.1815 - accuracy:
0.5713 - val_loss: 2.1546 - val_accuracy: 0.4204
Epoch 8/20
75/75 [==============================] - 3s 43ms/step - loss: 1.1028 - accuracy:
0.6047 - val_loss: 1.3870 - val_accuracy: 0.5336
Epoch 9/20
75/75 [==============================] - 3s 43ms/step - loss: 1.0428 - accuracy:
0.6215 - val_loss: 1.3632 - val_accuracy: 0.5500
Epoch 10/20
75/75 [==============================] - 3s 44ms/step - loss: 0.9860 - accuracy:
0.6485 - val_loss: 1.3977 - val_accuracy: 0.5584
Epoch 11/20
75/75 [==============================] - 3s 43ms/step - loss: 0.9270 - accuracy:
0.6663 - val_loss: 1.3836 - val_accuracy: 0.5704
Epoch 12/20
75/75 [==============================] - 3s 44ms/step - loss: 0.8730 - accuracy:
0.6976 - val_loss: 1.3521 - val_accuracy: 0.5664
Epoch 13/20
75/75 [==============================] - 3s 43ms/step - loss: 0.8448 - accuracy:
0.7031 - val_loss: 1.4237 - val_accuracy: 0.5668
Epoch 14/20
75/75 [==============================] - 3s 42ms/step - loss: 0.7414 - accuracy:
0.7373 - val_loss: 1.4741 - val_accuracy: 0.5660
Epoch 15/20
75/75 [==============================] - 3s 42ms/step - loss: 0.7217 - accuracy:
0.7397 - val_loss: 1.7016 - val_accuracy: 0.5320
Epoch 16/20
75/75 [==============================] - 3s 43ms/step - loss: 0.6792 - accuracy:
0.7592 - val_loss: 1.6758 - val_accuracy: 0.5508
Epoch 17/20
75/75 [==============================] - 3s 43ms/step - loss: 0.6296 - accuracy:
0.7753 - val_loss: 1.6354 - val_accuracy: 0.5756
Epoch 18/20
75/75 [==============================] - 3s 42ms/step - loss: 0.6038 - accuracy:
0.7848 - val_loss: 1.7425 - val_accuracy: 0.5588
Epoch 19/20
75/75 [==============================] - 3s 43ms/step - loss: 0.5224 - accuracy:
0.8157 - val_loss: 1.6230 - val_accuracy: 0.5668
```

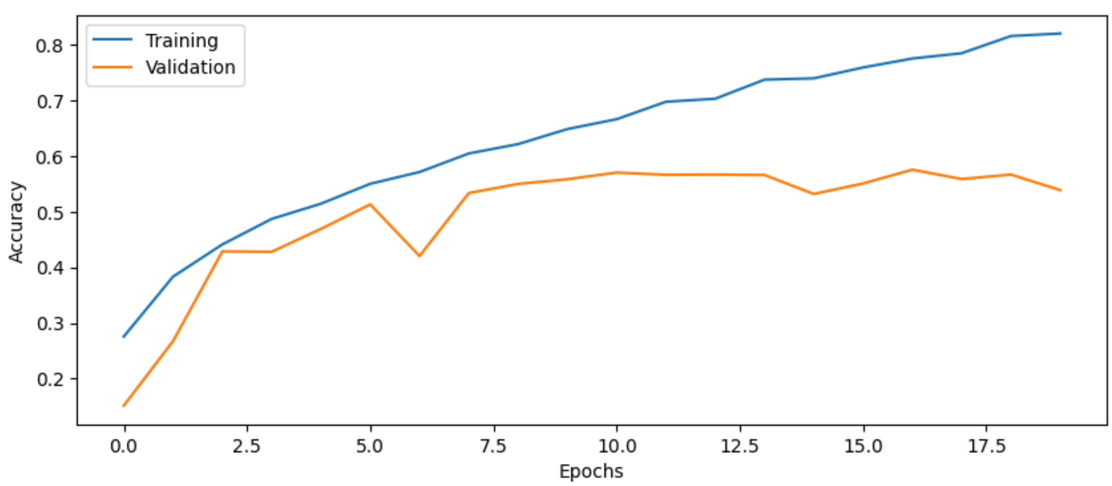
```
Epoch 20/20
75/75 [==============================] - 3s 43ms/step - loss: 0.5098 - accuracy:
0.8203 - val_loss: 1.8491 - val_accuracy: 0.5388
```

[23]:
```
# 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 5ms/step - loss: 1.9404 - accuracy:
0.5450
Test loss: 1.9404
Test accuracy: 0.5450
```

[24]:
```
# Plot the history from the training run
plot_results(history4)
```





Answer: Question 18

- The test accuracy went from 0.5430 to 0.5450 when we used dropout, not a big improvement.

Question 19

- Other types of regularization could be more trainingdata, early stopping, data augmentation. We could add L2 regularization for the convolutional layers by adding the arguments "kernel_regularizer = l2(*lambda1*)" and "bias_regularizer = l2(*lambda2*)". *lambda1* and *lambda2* is how much we want to regularize.

1.19 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?

1.20 Your best config

[25]:
```
# Setup some training parameters
batch_size = 100
epochs = 30
input_shape = (32,32,3)

# Build model
model5 = build_CNN(input_shape = input_shape, n_conv_layers = 5, n_dense_layers␣
↪= 3, n_nodes=64, use_dropout = True)

# Train the model  using training data and validation data
history5 = model5.fit(Xtrain, Ytrain, validation_data = (Xval, Yval),␣
↪batch_size = batch_size, epochs = epochs)
```

```
Epoch 1/30
75/75 [==============================] - 6s 47ms/step - loss: 2.2969 - accuracy:
0.2075 - val_loss: 2.0631 - val_accuracy: 0.2140
Epoch 2/30
75/75 [==============================] - 3s 44ms/step - loss: 1.8692 - accuracy:
0.2843 - val_loss: 2.0969 - val_accuracy: 0.2236
Epoch 3/30
75/75 [==============================] - 3s 44ms/step - loss: 1.7496 - accuracy:
0.3361 - val_loss: 1.9414 - val_accuracy: 0.3128
Epoch 4/30
75/75 [==============================] - 3s 43ms/step - loss: 1.6560 - accuracy:
0.3736 - val_loss: 1.7605 - val_accuracy: 0.3828
```

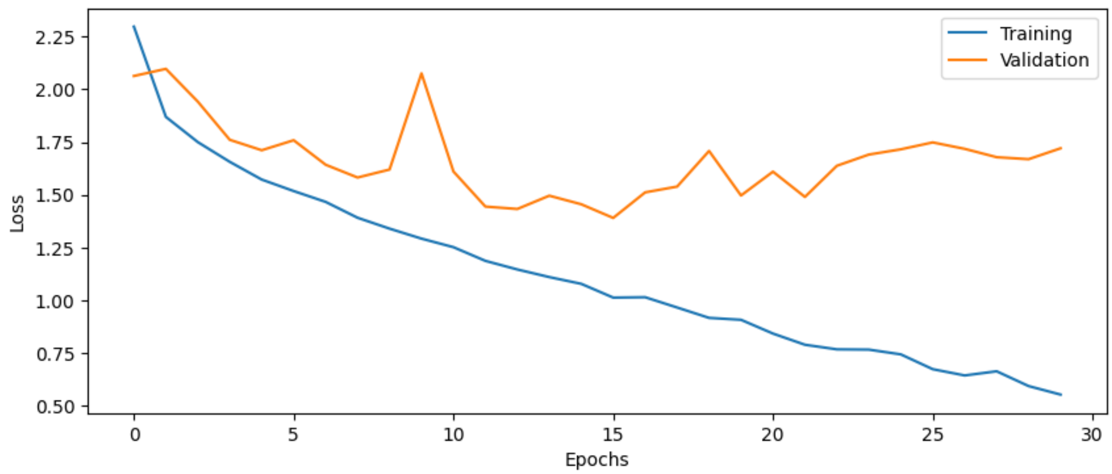
```
Epoch 5/30
75/75 [==============================] - 4s 48ms/step - loss: 1.5720 - accuracy:
0.4135 - val_loss: 1.7111 - val_accuracy: 0.4120
Epoch 6/30
75/75 [==============================] - 3s 45ms/step - loss: 1.5176 - accuracy:
0.4325 - val_loss: 1.7585 - val_accuracy: 0.3864
Epoch 7/30
75/75 [==============================] - 3s 46ms/step - loss: 1.4662 - accuracy:
0.4576 - val_loss: 1.6424 - val_accuracy: 0.4384
Epoch 8/30
75/75 [==============================] - 3s 43ms/step - loss: 1.3909 - accuracy:
0.4877 - val_loss: 1.5816 - val_accuracy: 0.4672
Epoch 9/30
75/75 [==============================] - 3s 46ms/step - loss: 1.3394 - accuracy:
0.5087 - val_loss: 1.6196 - val_accuracy: 0.4512
Epoch 10/30
75/75 [==============================] - 3s 45ms/step - loss: 1.2920 - accuracy:
0.5311 - val_loss: 2.0754 - val_accuracy: 0.3980
Epoch 11/30
75/75 [==============================] - 3s 44ms/step - loss: 1.2514 - accuracy:
0.5475 - val_loss: 1.6102 - val_accuracy: 0.4792
Epoch 12/30
75/75 [==============================] - 3s 45ms/step - loss: 1.1865 - accuracy:
0.5703 - val_loss: 1.4439 - val_accuracy: 0.5316
Epoch 13/30
75/75 [==============================] - 3s 45ms/step - loss: 1.1457 - accuracy:
0.5903 - val_loss: 1.4326 - val_accuracy: 0.5104
Epoch 14/30
75/75 [==============================] - 4s 52ms/step - loss: 1.1097 - accuracy:
0.6116 - val_loss: 1.4955 - val_accuracy: 0.5296
Epoch 15/30
75/75 [==============================] - 3s 47ms/step - loss: 1.0777 - accuracy:
0.6216 - val_loss: 1.4549 - val_accuracy: 0.5328
Epoch 16/30
75/75 [==============================] - 4s 54ms/step - loss: 1.0124 - accuracy:
0.6487 - val_loss: 1.3902 - val_accuracy: 0.5332
Epoch 17/30
75/75 [==============================] - 3s 44ms/step - loss: 1.0141 - accuracy:
0.6557 - val_loss: 1.5113 - val_accuracy: 0.5224
Epoch 18/30
75/75 [==============================] - 3s 43ms/step - loss: 0.9654 - accuracy:
0.6688 - val_loss: 1.5387 - val_accuracy: 0.5400
Epoch 19/30
75/75 [==============================] - 3s 46ms/step - loss: 0.9162 - accuracy:
0.6779 - val_loss: 1.7076 - val_accuracy: 0.5036
Epoch 20/30
75/75 [==============================] - 3s 45ms/step - loss: 0.9071 - accuracy:
0.6927 - val_loss: 1.4965 - val_accuracy: 0.5484
```

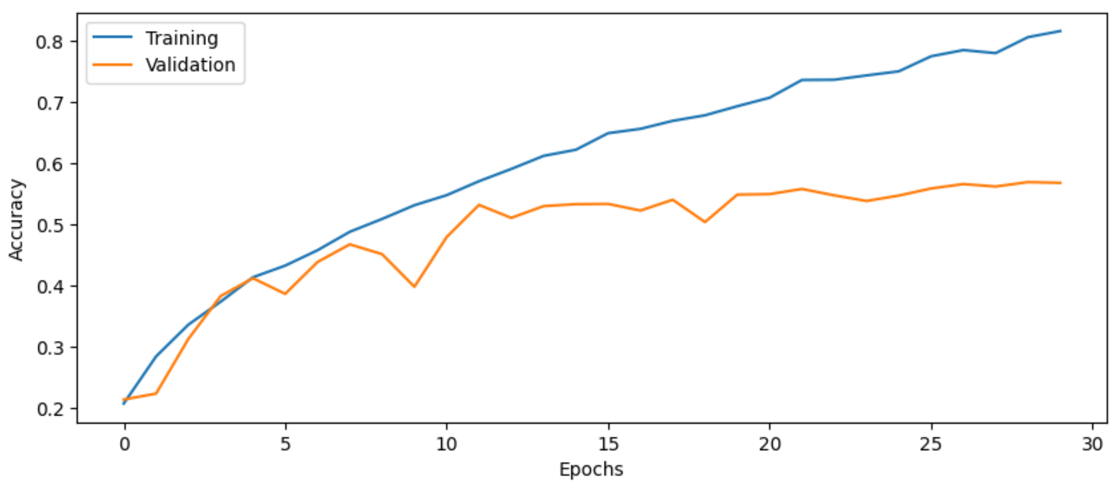
```
Epoch 21/30
75/75 [==============================] - 3s 45ms/step - loss: 0.8419 - accuracy:
0.7065 - val_loss: 1.6094 - val_accuracy: 0.5492
Epoch 22/30
75/75 [==============================] - 3s 44ms/step - loss: 0.7889 - accuracy:
0.7355 - val_loss: 1.4897 - val_accuracy: 0.5576
Epoch 23/30
75/75 [==============================] - 3s 45ms/step - loss: 0.7670 - accuracy:
0.7359 - val_loss: 1.6373 - val_accuracy: 0.5472
Epoch 24/30
75/75 [==============================] - 3s 44ms/step - loss: 0.7658 - accuracy:
0.7429 - val_loss: 1.6905 - val_accuracy: 0.5380
Epoch 25/30
75/75 [==============================] - 3s 44ms/step - loss: 0.7433 - accuracy:
0.7496 - val_loss: 1.7154 - val_accuracy: 0.5468
Epoch 26/30
75/75 [==============================] - 3s 45ms/step - loss: 0.6730 - accuracy:
0.7741 - val_loss: 1.7482 - val_accuracy: 0.5584
Epoch 27/30
75/75 [==============================] - 3s 46ms/step - loss: 0.6433 - accuracy:
0.7843 - val_loss: 1.7177 - val_accuracy: 0.5656
Epoch 28/30
75/75 [==============================] - 3s 45ms/step - loss: 0.6630 - accuracy:
0.7793 - val_loss: 1.6781 - val_accuracy: 0.5616
Epoch 29/30
75/75 [==============================] - 3s 46ms/step - loss: 0.5929 - accuracy:
0.8053 - val_loss: 1.6686 - val_accuracy: 0.5688
Epoch 30/30
75/75 [==============================] - 3s 45ms/step - loss: 0.5525 - accuracy:
0.8153 - val_loss: 1.7202 - val_accuracy: 0.5676
```

[26]:
```python
# Evaluate the trained model on test set, not used in training or validation
score = model5.evaluate(Xtest, Ytest)
print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])
```

```
63/63 [==============================] - 1s 5ms/step - loss: 1.7701 - accuracy:
0.5515
Test loss: 1.7701
Test accuracy: 0.5515
```

[27]:
```python
# Plot the history from the training run
plot_results(history5)
```





Answer: Question 20

- The test accuracy is 0.5515. We used batch_size = 100, epochs = 30, n_conv_layers = 5, n_dense_layers = 3n_nodes=64 and use_dropout = True.

1.21 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.

```
[28]: def myrotate(images):

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

          return images_rot
```

```
[29]: # 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()
```


```
[30]: # 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 [==============================] - 0s 5ms/step - loss: 4.2155 - accuracy:
0.2160
Test loss: 4.2155
Test accuracy: 0.2160
```

Answer: **Question 21**

- The accuracy is around 22 % compared to 55 % when we did not use rotated images. This is because the model is trained on "correctly" rotated images.

1.22 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…

[31]:
```
# Get all 60 000 training images again. ImageDataGenerator manages validation␣
↪data on its own
(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)
```

[32]:
```
# 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=30,
    horizontal_flip=True,
    vertical_flip=True,
    validation_split=0.20)

# Setup a flow for training data, assume that we can fit all images into CPU␣
↪memory
train_generator = datagen.flow(Xtrain, Ytrain, subset="training")
```

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

1.23 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: Question 22

- We would change the add the argument 'batch_size' in the .flow function to a lower value that fits in memory. This will take longer time to train the model compared to if all images would fit in memory. Alternatively, we can change the flow function to flow_from_directory, where we instead stream the data from directory which also increases training time.

[33]:
```python
# 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()
```

1.24 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?

[34]:
```
# Setup some training parameters
batch_size = 100
epochs = 200
input_shape = (32, 32, 3)

# Build model (your best config)
model6 = build_CNN(input_shape = input_shape, n_conv_layers = 5, n_dense_layers␣
 ↪= 3, n_nodes=64, use_dropout = True)
validation_split = 0.2

# Train the model using on the fly augmentation
history6 = model6.fit(train_generator,
                      validation_data = validation_generator,
                      batch_size=batch_size,
                      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 [==============================] - 5s 25ms/step - loss: 2.5719 - accuracy:
0.1496 - val_loss: 2.6504 - val_accuracy: 0.2047
Epoch 2/200
80/80 [==============================] - 2s 23ms/step - loss: 2.1225 - accuracy:
0.1965 - val_loss: 2.1171 - val_accuracy: 0.1828
Epoch 3/200
```

```
80/80 [==============================] - 2s 30ms/step - loss: 2.0357 - accuracy:
0.2398 - val_loss: 2.0927 - val_accuracy: 0.1859
Epoch 4/200
80/80 [==============================] - 2s 24ms/step - loss: 2.0061 - accuracy:
0.2313 - val_loss: 1.8280 - val_accuracy: 0.3109
Epoch 5/200
80/80 [==============================] - 2s 23ms/step - loss: 1.9821 - accuracy:
0.2363 - val_loss: 1.8274 - val_accuracy: 0.2578
Epoch 6/200
80/80 [==============================] - 2s 23ms/step - loss: 1.9106 - accuracy:
0.2707 - val_loss: 1.9699 - val_accuracy: 0.2641
Epoch 7/200
80/80 [==============================] - 2s 22ms/step - loss: 1.9124 - accuracy:
0.2621 - val_loss: 1.7777 - val_accuracy: 0.2828
Epoch 8/200
80/80 [==============================] - 2s 23ms/step - loss: 1.9244 - accuracy:
0.2594 - val_loss: 1.9075 - val_accuracy: 0.2703
Epoch 9/200
80/80 [==============================] - 2s 23ms/step - loss: 1.8898 - accuracy:
0.2727 - val_loss: 1.8638 - val_accuracy: 0.3016
Epoch 10/200
80/80 [==============================] - 2s 22ms/step - loss: 1.8929 - accuracy:
0.2738 - val_loss: 1.9061 - val_accuracy: 0.2562
Epoch 11/200
80/80 [==============================] - 2s 23ms/step - loss: 1.8946 - accuracy:
0.2926 - val_loss: 1.8956 - val_accuracy: 0.3250
Epoch 12/200
80/80 [==============================] - 2s 22ms/step - loss: 1.8796 - accuracy:
0.2711 - val_loss: 1.8484 - val_accuracy: 0.2703
Epoch 13/200
80/80 [==============================] - 2s 24ms/step - loss: 1.8518 - accuracy:
0.2867 - val_loss: 1.8109 - val_accuracy: 0.3172
Epoch 14/200
80/80 [==============================] - 2s 23ms/step - loss: 1.8586 - accuracy:
0.2812 - val_loss: 1.8060 - val_accuracy: 0.2922
Epoch 15/200
80/80 [==============================] - 2s 22ms/step - loss: 1.8180 - accuracy:
0.2988 - val_loss: 1.8409 - val_accuracy: 0.3125
Epoch 16/200
80/80 [==============================] - 2s 24ms/step - loss: 1.8477 - accuracy:
0.2977 - val_loss: 1.7702 - val_accuracy: 0.3156
Epoch 17/200
80/80 [==============================] - 2s 23ms/step - loss: 1.8477 - accuracy:
0.2949 - val_loss: 2.0450 - val_accuracy: 0.2609
Epoch 18/200
80/80 [==============================] - 2s 22ms/step - loss: 1.8430 - accuracy:
0.2988 - val_loss: 1.6958 - val_accuracy: 0.3719
Epoch 19/200
```

```
80/80 [==============================] - 2s 23ms/step - loss: 1.8154 - accuracy:
0.3082 - val_loss: 1.7429 - val_accuracy: 0.3344
Epoch 20/200
80/80 [==============================] - 2s 22ms/step - loss: 1.8150 - accuracy:
0.3207 - val_loss: 1.7317 - val_accuracy: 0.3672
Epoch 21/200
80/80 [==============================] - 2s 24ms/step - loss: 1.8200 - accuracy:
0.3230 - val_loss: 1.6855 - val_accuracy: 0.3469
Epoch 22/200
80/80 [==============================] - 2s 24ms/step - loss: 1.7777 - accuracy:
0.3262 - val_loss: 1.7228 - val_accuracy: 0.3547
Epoch 23/200
80/80 [==============================] - 2s 27ms/step - loss: 1.7945 - accuracy:
0.3141 - val_loss: 1.7418 - val_accuracy: 0.3266
Epoch 24/200
80/80 [==============================] - 2s 27ms/step - loss: 1.7812 - accuracy:
0.3328 - val_loss: 1.6546 - val_accuracy: 0.3734
Epoch 25/200
80/80 [==============================] - 2s 28ms/step - loss: 1.7558 - accuracy:
0.3430 - val_loss: 1.6522 - val_accuracy: 0.3703
Epoch 26/200
80/80 [==============================] - 2s 28ms/step - loss: 1.7677 - accuracy:
0.3270 - val_loss: 1.7055 - val_accuracy: 0.3625
Epoch 27/200
80/80 [==============================] - 2s 27ms/step - loss: 1.7713 - accuracy:
0.3320 - val_loss: 1.7042 - val_accuracy: 0.3500
Epoch 28/200
80/80 [==============================] - 2s 26ms/step - loss: 1.7732 - accuracy:
0.3313 - val_loss: 1.7031 - val_accuracy: 0.3516
Epoch 29/200
80/80 [==============================] - 2s 27ms/step - loss: 1.7450 - accuracy:
0.3375 - val_loss: 1.6509 - val_accuracy: 0.3875
Epoch 30/200
80/80 [==============================] - 2s 23ms/step - loss: 1.7568 - accuracy:
0.3516 - val_loss: 1.5720 - val_accuracy: 0.3828
Epoch 31/200
80/80 [==============================] - 2s 23ms/step - loss: 1.7317 - accuracy:
0.3586 - val_loss: 1.5788 - val_accuracy: 0.4047
Epoch 32/200
80/80 [==============================] - 2s 22ms/step - loss: 1.7207 - accuracy:
0.3527 - val_loss: 1.6737 - val_accuracy: 0.3828
Epoch 33/200
80/80 [==============================] - 2s 23ms/step - loss: 1.7356 - accuracy:
0.3535 - val_loss: 1.6269 - val_accuracy: 0.4016
Epoch 34/200
80/80 [==============================] - 2s 22ms/step - loss: 1.7115 - accuracy:
0.3570 - val_loss: 1.6067 - val_accuracy: 0.3609
Epoch 35/200
```

```
80/80 [==============================] - 2s 24ms/step - loss: 1.6906 - accuracy:
0.3734 - val_loss: 1.6340 - val_accuracy: 0.4016
Epoch 36/200
80/80 [==============================] - 2s 23ms/step - loss: 1.6749 - accuracy:
0.3812 - val_loss: 1.5965 - val_accuracy: 0.3672
Epoch 37/200
80/80 [==============================] - 2s 23ms/step - loss: 1.7062 - accuracy:
0.3684 - val_loss: 1.7209 - val_accuracy: 0.3531
Epoch 38/200
80/80 [==============================] - 2s 23ms/step - loss: 1.6716 - accuracy:
0.3652 - val_loss: 1.5728 - val_accuracy: 0.4172
Epoch 39/200
80/80 [==============================] - 2s 23ms/step - loss: 1.6846 - accuracy:
0.3613 - val_loss: 1.6053 - val_accuracy: 0.3734
Epoch 40/200
80/80 [==============================] - 2s 23ms/step - loss: 1.6758 - accuracy:
0.3801 - val_loss: 1.6860 - val_accuracy: 0.3734
Epoch 41/200
80/80 [==============================] - 2s 30ms/step - loss: 1.6734 - accuracy:
0.3695 - val_loss: 1.4884 - val_accuracy: 0.4328
Epoch 42/200
80/80 [==============================] - 2s 23ms/step - loss: 1.6925 - accuracy:
0.3867 - val_loss: 1.6089 - val_accuracy: 0.3922
Epoch 43/200
80/80 [==============================] - 2s 23ms/step - loss: 1.6239 - accuracy:
0.3906 - val_loss: 1.5806 - val_accuracy: 0.3719
Epoch 44/200
80/80 [==============================] - 2s 24ms/step - loss: 1.6514 - accuracy:
0.3871 - val_loss: 1.5351 - val_accuracy: 0.4203
Epoch 45/200
80/80 [==============================] - 2s 23ms/step - loss: 1.6885 - accuracy:
0.3762 - val_loss: 1.4592 - val_accuracy: 0.4453
Epoch 46/200
80/80 [==============================] - 2s 23ms/step - loss: 1.6678 - accuracy:
0.3777 - val_loss: 1.5653 - val_accuracy: 0.4328
Epoch 47/200
80/80 [==============================] - 2s 23ms/step - loss: 1.6571 - accuracy:
0.3828 - val_loss: 1.5143 - val_accuracy: 0.4313
Epoch 48/200
80/80 [==============================] - 2s 23ms/step - loss: 1.6262 - accuracy:
0.3773 - val_loss: 1.5960 - val_accuracy: 0.3547
Epoch 49/200
80/80 [==============================] - 2s 27ms/step - loss: 1.6427 - accuracy:
0.3953 - val_loss: 1.4732 - val_accuracy: 0.4359
Epoch 50/200
80/80 [==============================] - 2s 30ms/step - loss: 1.6428 - accuracy:
0.3711 - val_loss: 1.5927 - val_accuracy: 0.4250
Epoch 51/200
```

```
80/80 [==============================] - 2s 28ms/step - loss: 1.6346 - accuracy:
0.3988 - val_loss: 1.4011 - val_accuracy: 0.4500
Epoch 52/200
80/80 [==============================] - 2s 28ms/step - loss: 1.6569 - accuracy:
0.3805 - val_loss: 1.4659 - val_accuracy: 0.4422
Epoch 53/200
80/80 [==============================] - 2s 29ms/step - loss: 1.6348 - accuracy:
0.3879 - val_loss: 1.5316 - val_accuracy: 0.4297
Epoch 54/200
80/80 [==============================] - 2s 28ms/step - loss: 1.6264 - accuracy:
0.4012 - val_loss: 1.5330 - val_accuracy: 0.4078
Epoch 55/200
80/80 [==============================] - 2s 28ms/step - loss: 1.6539 - accuracy:
0.3930 - val_loss: 1.5406 - val_accuracy: 0.4344
Epoch 56/200
80/80 [==============================] - 2s 28ms/step - loss: 1.6178 - accuracy:
0.4027 - val_loss: 1.5125 - val_accuracy: 0.4469
Epoch 57/200
80/80 [==============================] - 2s 28ms/step - loss: 1.6125 - accuracy:
0.4051 - val_loss: 1.4778 - val_accuracy: 0.4313
Epoch 58/200
80/80 [==============================] - 2s 29ms/step - loss: 1.6460 - accuracy:
0.4043 - val_loss: 1.5902 - val_accuracy: 0.3969
Epoch 59/200
80/80 [==============================] - 2s 30ms/step - loss: 1.6103 - accuracy:
0.4027 - val_loss: 1.4689 - val_accuracy: 0.4453
Epoch 60/200
80/80 [==============================] - 2s 29ms/step - loss: 1.6196 - accuracy:
0.4047 - val_loss: 1.5127 - val_accuracy: 0.4313
Epoch 61/200
80/80 [==============================] - 2s 29ms/step - loss: 1.5633 - accuracy:
0.4238 - val_loss: 1.6170 - val_accuracy: 0.4297
Epoch 62/200
80/80 [==============================] - 2s 29ms/step - loss: 1.6442 - accuracy:
0.3926 - val_loss: 1.4309 - val_accuracy: 0.4563
Epoch 63/200
80/80 [==============================] - 2s 31ms/step - loss: 1.6082 - accuracy:
0.4203 - val_loss: 1.5363 - val_accuracy: 0.4047
Epoch 64/200
80/80 [==============================] - 3s 31ms/step - loss: 1.5915 - accuracy:
0.4137 - val_loss: 1.4566 - val_accuracy: 0.4406
Epoch 65/200
80/80 [==============================] - 2s 29ms/step - loss: 1.5805 - accuracy:
0.4168 - val_loss: 1.4550 - val_accuracy: 0.4641
Epoch 66/200
80/80 [==============================] - 3s 33ms/step - loss: 1.5564 - accuracy:
0.4320 - val_loss: 1.4997 - val_accuracy: 0.4313
Epoch 67/200
```

```
80/80 [==============================] - 2s 29ms/step - loss: 1.5536 - accuracy:
0.4359 - val_loss: 1.5111 - val_accuracy: 0.4422
Epoch 68/200
80/80 [==============================] - 2s 31ms/step - loss: 1.5658 - accuracy:
0.4344 - val_loss: 1.5171 - val_accuracy: 0.4266
Epoch 69/200
80/80 [==============================] - 2s 30ms/step - loss: 1.5711 - accuracy:
0.4211 - val_loss: 1.4416 - val_accuracy: 0.4875
Epoch 70/200
80/80 [==============================] - 3s 32ms/step - loss: 1.5827 - accuracy:
0.4367 - val_loss: 1.5271 - val_accuracy: 0.4313
Epoch 71/200
80/80 [==============================] - 2s 28ms/step - loss: 1.5826 - accuracy:
0.3957 - val_loss: 1.4214 - val_accuracy: 0.4766
Epoch 72/200
80/80 [==============================] - 3s 41ms/step - loss: 1.5330 - accuracy:
0.4441 - val_loss: 1.4181 - val_accuracy: 0.4437
Epoch 73/200
80/80 [==============================] - 2s 30ms/step - loss: 1.5901 - accuracy:
0.4059 - val_loss: 1.5003 - val_accuracy: 0.4609
Epoch 74/200
80/80 [==============================] - 2s 30ms/step - loss: 1.5644 - accuracy:
0.4219 - val_loss: 1.5562 - val_accuracy: 0.3969
Epoch 75/200
80/80 [==============================] - 2s 30ms/step - loss: 1.5745 - accuracy:
0.4430 - val_loss: 1.5377 - val_accuracy: 0.4359
Epoch 76/200
80/80 [==============================] - 2s 26ms/step - loss: 1.5602 - accuracy:
0.4344 - val_loss: 1.4147 - val_accuracy: 0.4969
Epoch 77/200
80/80 [==============================] - 2s 26ms/step - loss: 1.5689 - accuracy:
0.4379 - val_loss: 1.3538 - val_accuracy: 0.4812
Epoch 78/200
80/80 [==============================] - 2s 28ms/step - loss: 1.5552 - accuracy:
0.4406 - val_loss: 1.4099 - val_accuracy: 0.4656
Epoch 79/200
80/80 [==============================] - 2s 26ms/step - loss: 1.5512 - accuracy:
0.4445 - val_loss: 1.3600 - val_accuracy: 0.4984
Epoch 80/200
80/80 [==============================] - 2s 24ms/step - loss: 1.5221 - accuracy:
0.4465 - val_loss: 1.4482 - val_accuracy: 0.4672
Epoch 81/200
80/80 [==============================] - 2s 26ms/step - loss: 1.5178 - accuracy:
0.4609 - val_loss: 1.4980 - val_accuracy: 0.4297
Epoch 82/200
80/80 [==============================] - 2s 28ms/step - loss: 1.5345 - accuracy:
0.4316 - val_loss: 1.4133 - val_accuracy: 0.4766
Epoch 83/200
```

```
80/80 [==============================] - 2s 29ms/step - loss: 1.5081 - accuracy:
0.4512 - val_loss: 1.4239 - val_accuracy: 0.4688
Epoch 84/200
80/80 [==============================] - 2s 28ms/step - loss: 1.5363 - accuracy:
0.4551 - val_loss: 1.3846 - val_accuracy: 0.4609
Epoch 85/200
80/80 [==============================] - 2s 29ms/step - loss: 1.5048 - accuracy:
0.4613 - val_loss: 1.3477 - val_accuracy: 0.5172
Epoch 86/200
80/80 [==============================] - 2s 29ms/step - loss: 1.5266 - accuracy:
0.4563 - val_loss: 1.3923 - val_accuracy: 0.4859
Epoch 87/200
80/80 [==============================] - 2s 27ms/step - loss: 1.4913 - accuracy:
0.4695 - val_loss: 1.3894 - val_accuracy: 0.4656
Epoch 88/200
80/80 [==============================] - 2s 28ms/step - loss: 1.5361 - accuracy:
0.4500 - val_loss: 1.4444 - val_accuracy: 0.4719
Epoch 89/200
80/80 [==============================] - 3s 33ms/step - loss: 1.4910 - accuracy:
0.4617 - val_loss: 1.5287 - val_accuracy: 0.4437
Epoch 90/200
80/80 [==============================] - 2s 29ms/step - loss: 1.5085 - accuracy:
0.4641 - val_loss: 1.4469 - val_accuracy: 0.5016
Epoch 91/200
80/80 [==============================] - 2s 30ms/step - loss: 1.4660 - accuracy:
0.4727 - val_loss: 1.3725 - val_accuracy: 0.5047
Epoch 92/200
80/80 [==============================] - 2s 30ms/step - loss: 1.4774 - accuracy:
0.4816 - val_loss: 1.4309 - val_accuracy: 0.4672
Epoch 93/200
80/80 [==============================] - 2s 30ms/step - loss: 1.5043 - accuracy:
0.4684 - val_loss: 1.4455 - val_accuracy: 0.4922
Epoch 94/200
80/80 [==============================] - 2s 28ms/step - loss: 1.5192 - accuracy:
0.4648 - val_loss: 1.4502 - val_accuracy: 0.4531
Epoch 95/200
80/80 [==============================] - 2s 26ms/step - loss: 1.4701 - accuracy:
0.4773 - val_loss: 1.5040 - val_accuracy: 0.4500
Epoch 96/200
80/80 [==============================] - 2s 31ms/step - loss: 1.4855 - accuracy:
0.4684 - val_loss: 1.3521 - val_accuracy: 0.5047
Epoch 97/200
80/80 [==============================] - 3s 33ms/step - loss: 1.5051 - accuracy:
0.4621 - val_loss: 1.3400 - val_accuracy: 0.4984
Epoch 98/200
80/80 [==============================] - 3s 31ms/step - loss: 1.4813 - accuracy:
0.4684 - val_loss: 1.4270 - val_accuracy: 0.4953
Epoch 99/200
```

```
80/80 [==============================] - 2s 30ms/step - loss: 1.5110 - accuracy:
0.4535 - val_loss: 1.4433 - val_accuracy: 0.4406
Epoch 100/200
80/80 [==============================] - 2s 29ms/step - loss: 1.4776 - accuracy:
0.4730 - val_loss: 1.3341 - val_accuracy: 0.4984
Epoch 101/200
80/80 [==============================] - 2s 27ms/step - loss: 1.4964 - accuracy:
0.4793 - val_loss: 1.4595 - val_accuracy: 0.4625
Epoch 102/200
80/80 [==============================] - 2s 27ms/step - loss: 1.4745 - accuracy:
0.4828 - val_loss: 1.3946 - val_accuracy: 0.4891
Epoch 103/200
80/80 [==============================] - 2s 26ms/step - loss: 1.4777 - accuracy:
0.4691 - val_loss: 1.3725 - val_accuracy: 0.5016
Epoch 104/200
80/80 [==============================] - 2s 27ms/step - loss: 1.4824 - accuracy:
0.4699 - val_loss: 1.3013 - val_accuracy: 0.5281
Epoch 105/200
80/80 [==============================] - 2s 25ms/step - loss: 1.4725 - accuracy:
0.4824 - val_loss: 1.3824 - val_accuracy: 0.4969
Epoch 106/200
80/80 [==============================] - 2s 21ms/step - loss: 1.4337 - accuracy:
0.4988 - val_loss: 1.3846 - val_accuracy: 0.4781
Epoch 107/200
80/80 [==============================] - 2s 22ms/step - loss: 1.4592 - accuracy:
0.4793 - val_loss: 1.4262 - val_accuracy: 0.4828
Epoch 108/200
80/80 [==============================] - 2s 24ms/step - loss: 1.4664 - accuracy:
0.4938 - val_loss: 1.3979 - val_accuracy: 0.4797
Epoch 109/200
80/80 [==============================] - 2s 21ms/step - loss: 1.4606 - accuracy:
0.4855 - val_loss: 1.3815 - val_accuracy: 0.4906
Epoch 110/200
80/80 [==============================] - 2s 24ms/step - loss: 1.4768 - accuracy:
0.4684 - val_loss: 1.3234 - val_accuracy: 0.4969
Epoch 111/200
80/80 [==============================] - 2s 21ms/step - loss: 1.4704 - accuracy:
0.4727 - val_loss: 1.3074 - val_accuracy: 0.5219
Epoch 112/200
80/80 [==============================] - 2s 22ms/step - loss: 1.4529 - accuracy:
0.4777 - val_loss: 1.3959 - val_accuracy: 0.4766
Epoch 113/200
80/80 [==============================] - 2s 22ms/step - loss: 1.4570 - accuracy:
0.4680 - val_loss: 1.3228 - val_accuracy: 0.5031
Epoch 114/200
80/80 [==============================] - 2s 22ms/step - loss: 1.4191 - accuracy:
0.5008 - val_loss: 1.3864 - val_accuracy: 0.4922
Epoch 115/200
```

```
80/80 [==============================] - 2s 20ms/step - loss: 1.4187 - accuracy:
0.4953 - val_loss: 1.3740 - val_accuracy: 0.4891
Epoch 116/200
80/80 [==============================] - 2s 21ms/step - loss: 1.4506 - accuracy:
0.4715 - val_loss: 1.3639 - val_accuracy: 0.4938
Epoch 117/200
80/80 [==============================] - 2s 22ms/step - loss: 1.4354 - accuracy:
0.5012 - val_loss: 1.3877 - val_accuracy: 0.4812
Epoch 118/200
80/80 [==============================] - 2s 22ms/step - loss: 1.4266 - accuracy:
0.4969 - val_loss: 1.3589 - val_accuracy: 0.5406
Epoch 119/200
80/80 [==============================] - 2s 22ms/step - loss: 1.4755 - accuracy:
0.4754 - val_loss: 1.2915 - val_accuracy: 0.5156
Epoch 120/200
80/80 [==============================] - 2s 26ms/step - loss: 1.4273 - accuracy:
0.4984 - val_loss: 1.2950 - val_accuracy: 0.5188
Epoch 121/200
80/80 [==============================] - 2s 22ms/step - loss: 1.4178 - accuracy:
0.4883 - val_loss: 1.3353 - val_accuracy: 0.5109
Epoch 122/200
80/80 [==============================] - 2s 23ms/step - loss: 1.4390 - accuracy:
0.4855 - val_loss: 1.3263 - val_accuracy: 0.5297
Epoch 123/200
80/80 [==============================] - 2s 22ms/step - loss: 1.4044 - accuracy:
0.5063 - val_loss: 1.2598 - val_accuracy: 0.5422
Epoch 124/200
80/80 [==============================] - 2s 22ms/step - loss: 1.4349 - accuracy:
0.5066 - val_loss: 1.4459 - val_accuracy: 0.4953
Epoch 125/200
80/80 [==============================] - 2s 22ms/step - loss: 1.4129 - accuracy:
0.5117 - val_loss: 1.2680 - val_accuracy: 0.5422
Epoch 126/200
80/80 [==============================] - 2s 22ms/step - loss: 1.4206 - accuracy:
0.5059 - val_loss: 1.3101 - val_accuracy: 0.5250
Epoch 127/200
80/80 [==============================] - 2s 21ms/step - loss: 1.4019 - accuracy:
0.5063 - val_loss: 1.3917 - val_accuracy: 0.5094
Epoch 128/200
80/80 [==============================] - 2s 23ms/step - loss: 1.4025 - accuracy:
0.5078 - val_loss: 1.3558 - val_accuracy: 0.5156
Epoch 129/200
80/80 [==============================] - 2s 21ms/step - loss: 1.3894 - accuracy:
0.5074 - val_loss: 1.3684 - val_accuracy: 0.5078
Epoch 130/200
80/80 [==============================] - 2s 23ms/step - loss: 1.4108 - accuracy:
0.4996 - val_loss: 1.4262 - val_accuracy: 0.5344
Epoch 131/200
```

```
80/80 [==============================] - 2s 22ms/step - loss: 1.4259 - accuracy:
0.4891 - val_loss: 1.5046 - val_accuracy: 0.4625
Epoch 132/200
80/80 [==============================] - 2s 22ms/step - loss: 1.3780 - accuracy:
0.5094 - val_loss: 1.3006 - val_accuracy: 0.5125
Epoch 133/200
80/80 [==============================] - 2s 22ms/step - loss: 1.4615 - accuracy:
0.5043 - val_loss: 1.2909 - val_accuracy: 0.5094
Epoch 134/200
80/80 [==============================] - 2s 24ms/step - loss: 1.3620 - accuracy:
0.5191 - val_loss: 1.3392 - val_accuracy: 0.5203
Epoch 135/200
80/80 [==============================] - 2s 23ms/step - loss: 1.4164 - accuracy:
0.4984 - val_loss: 1.3798 - val_accuracy: 0.5188
Epoch 136/200
80/80 [==============================] - 2s 23ms/step - loss: 1.4359 - accuracy:
0.5031 - val_loss: 1.3622 - val_accuracy: 0.5234
Epoch 137/200
80/80 [==============================] - 2s 23ms/step - loss: 1.3866 - accuracy:
0.5168 - val_loss: 1.3349 - val_accuracy: 0.5453
Epoch 138/200
80/80 [==============================] - 2s 22ms/step - loss: 1.3890 - accuracy:
0.5066 - val_loss: 1.2545 - val_accuracy: 0.5453
Epoch 139/200
80/80 [==============================] - 2s 21ms/step - loss: 1.4322 - accuracy:
0.4898 - val_loss: 1.2561 - val_accuracy: 0.5437
Epoch 140/200
80/80 [==============================] - 2s 31ms/step - loss: 1.3639 - accuracy:
0.5164 - val_loss: 1.3052 - val_accuracy: 0.5453
Epoch 141/200
80/80 [==============================] - 2s 22ms/step - loss: 1.4013 - accuracy:
0.5031 - val_loss: 1.4084 - val_accuracy: 0.5000
Epoch 142/200
80/80 [==============================] - 2s 22ms/step - loss: 1.3746 - accuracy:
0.5109 - val_loss: 1.2181 - val_accuracy: 0.5625
Epoch 143/200
80/80 [==============================] - 2s 21ms/step - loss: 1.3881 - accuracy:
0.5176 - val_loss: 1.3625 - val_accuracy: 0.5047
Epoch 144/200
80/80 [==============================] - 2s 23ms/step - loss: 1.3787 - accuracy:
0.5195 - val_loss: 1.3518 - val_accuracy: 0.4969
Epoch 145/200
80/80 [==============================] - 2s 21ms/step - loss: 1.3820 - accuracy:
0.5105 - val_loss: 1.3259 - val_accuracy: 0.5078
Epoch 146/200
80/80 [==============================] - 2s 22ms/step - loss: 1.4033 - accuracy:
0.5172 - val_loss: 1.3113 - val_accuracy: 0.5219
Epoch 147/200
```

```
80/80 [==============================] - 2s 22ms/step - loss: 1.3652 - accuracy:
0.5270 - val_loss: 1.2695 - val_accuracy: 0.5406
Epoch 148/200
80/80 [==============================] - 2s 23ms/step - loss: 1.4056 - accuracy:
0.5156 - val_loss: 1.2942 - val_accuracy: 0.5250
Epoch 149/200
80/80 [==============================] - 2s 23ms/step - loss: 1.3506 - accuracy:
0.5340 - val_loss: 1.2851 - val_accuracy: 0.5047
Epoch 150/200
80/80 [==============================] - 2s 23ms/step - loss: 1.3668 - accuracy:
0.5219 - val_loss: 1.3386 - val_accuracy: 0.5297
Epoch 151/200
80/80 [==============================] - 2s 24ms/step - loss: 1.3816 - accuracy:
0.5156 - val_loss: 1.3273 - val_accuracy: 0.5125
Epoch 152/200
80/80 [==============================] - 2s 24ms/step - loss: 1.3790 - accuracy:
0.5227 - val_loss: 1.3809 - val_accuracy: 0.5156
Epoch 153/200
80/80 [==============================] - 2s 24ms/step - loss: 1.3559 - accuracy:
0.5234 - val_loss: 1.2875 - val_accuracy: 0.5344
Epoch 154/200
80/80 [==============================] - 2s 24ms/step - loss: 1.3521 - accuracy:
0.5238 - val_loss: 1.3118 - val_accuracy: 0.5406
Epoch 155/200
80/80 [==============================] - 2s 25ms/step - loss: 1.4045 - accuracy:
0.5098 - val_loss: 1.2588 - val_accuracy: 0.5422
Epoch 156/200
80/80 [==============================] - 2s 24ms/step - loss: 1.3509 - accuracy:
0.5176 - val_loss: 1.2892 - val_accuracy: 0.5234
Epoch 157/200
80/80 [==============================] - 2s 25ms/step - loss: 1.3553 - accuracy:
0.5188 - val_loss: 1.1524 - val_accuracy: 0.5734
Epoch 158/200
80/80 [==============================] - 2s 23ms/step - loss: 1.3428 - accuracy:
0.5414 - val_loss: 1.2071 - val_accuracy: 0.5578
Epoch 159/200
80/80 [==============================] - 2s 23ms/step - loss: 1.3303 - accuracy:
0.5324 - val_loss: 1.2239 - val_accuracy: 0.5734
Epoch 160/200
80/80 [==============================] - 2s 22ms/step - loss: 1.3594 - accuracy:
0.5172 - val_loss: 1.2804 - val_accuracy: 0.5484
Epoch 161/200
80/80 [==============================] - 2s 23ms/step - loss: 1.3251 - accuracy:
0.5328 - val_loss: 1.2669 - val_accuracy: 0.5500
Epoch 162/200
80/80 [==============================] - 2s 22ms/step - loss: 1.3459 - accuracy:
0.5336 - val_loss: 1.3835 - val_accuracy: 0.5031
Epoch 163/200
```

```
80/80 [==============================] - 2s 22ms/step - loss: 1.3971 - accuracy:
0.5215 - val_loss: 1.2872 - val_accuracy: 0.5562
Epoch 164/200
80/80 [==============================] - 2s 23ms/step - loss: 1.3340 - accuracy:
0.5379 - val_loss: 1.2360 - val_accuracy: 0.5375
Epoch 165/200
80/80 [==============================] - 2s 23ms/step - loss: 1.4033 - accuracy:
0.5105 - val_loss: 1.2144 - val_accuracy: 0.5484
Epoch 166/200
80/80 [==============================] - 2s 23ms/step - loss: 1.3463 - accuracy:
0.5297 - val_loss: 1.2656 - val_accuracy: 0.5453
Epoch 167/200
80/80 [==============================] - 2s 24ms/step - loss: 1.3567 - accuracy:
0.5191 - val_loss: 1.3308 - val_accuracy: 0.5250
Epoch 168/200
80/80 [==============================] - 2s 23ms/step - loss: 1.3661 - accuracy:
0.5172 - val_loss: 1.2816 - val_accuracy: 0.5188
Epoch 169/200
80/80 [==============================] - 2s 23ms/step - loss: 1.3526 - accuracy:
0.5379 - val_loss: 1.2449 - val_accuracy: 0.5203
Epoch 170/200
80/80 [==============================] - 2s 24ms/step - loss: 1.3554 - accuracy:
0.5242 - val_loss: 1.2757 - val_accuracy: 0.5453
Epoch 171/200
80/80 [==============================] - 2s 23ms/step - loss: 1.3358 - accuracy:
0.5367 - val_loss: 1.2722 - val_accuracy: 0.5422
Epoch 172/200
80/80 [==============================] - 2s 23ms/step - loss: 1.3365 - accuracy:
0.5348 - val_loss: 1.2632 - val_accuracy: 0.5484
Epoch 173/200
80/80 [==============================] - 2s 22ms/step - loss: 1.3399 - accuracy:
0.5340 - val_loss: 1.2738 - val_accuracy: 0.5484
Epoch 174/200
80/80 [==============================] - 2s 23ms/step - loss: 1.2929 - accuracy:
0.5520 - val_loss: 1.2760 - val_accuracy: 0.5344
Epoch 175/200
80/80 [==============================] - 2s 26ms/step - loss: 1.3418 - accuracy:
0.5223 - val_loss: 1.3093 - val_accuracy: 0.5297
Epoch 176/200
80/80 [==============================] - 2s 22ms/step - loss: 1.3328 - accuracy:
0.5371 - val_loss: 1.3024 - val_accuracy: 0.5359
Epoch 177/200
80/80 [==============================] - 2s 23ms/step - loss: 1.3444 - accuracy:
0.5238 - val_loss: 1.2091 - val_accuracy: 0.5641
Epoch 178/200
80/80 [==============================] - 2s 22ms/step - loss: 1.2908 - accuracy:
0.5578 - val_loss: 1.1799 - val_accuracy: 0.5484
Epoch 179/200
```

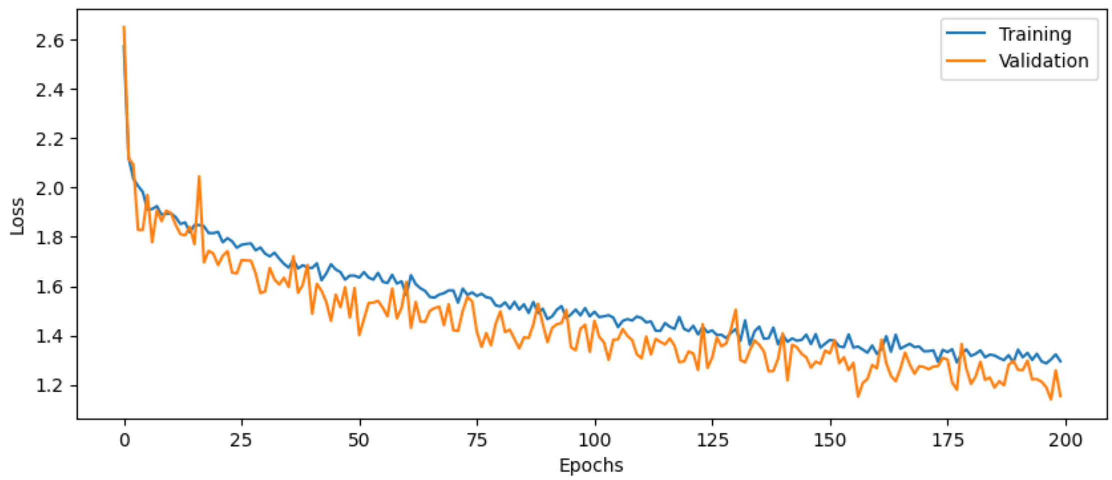
```
80/80 [==============================] - 2s 26ms/step - loss: 1.3260 - accuracy:
0.5395 - val_loss: 1.3654 - val_accuracy: 0.5078
Epoch 180/200
80/80 [==============================] - 2s 23ms/step - loss: 1.3437 - accuracy:
0.5336 - val_loss: 1.2664 - val_accuracy: 0.5625
Epoch 181/200
80/80 [==============================] - 2s 22ms/step - loss: 1.3170 - accuracy:
0.5391 - val_loss: 1.2036 - val_accuracy: 0.5578
Epoch 182/200
80/80 [==============================] - 2s 22ms/step - loss: 1.3258 - accuracy:
0.5340 - val_loss: 1.2343 - val_accuracy: 0.5656
Epoch 183/200
80/80 [==============================] - 2s 22ms/step - loss: 1.3397 - accuracy:
0.5437 - val_loss: 1.2935 - val_accuracy: 0.5250
Epoch 184/200
80/80 [==============================] - 2s 27ms/step - loss: 1.3114 - accuracy:
0.5492 - val_loss: 1.2200 - val_accuracy: 0.5484
Epoch 185/200
80/80 [==============================] - 2s 23ms/step - loss: 1.3223 - accuracy:
0.5402 - val_loss: 1.2296 - val_accuracy: 0.5641
Epoch 186/200
80/80 [==============================] - 2s 23ms/step - loss: 1.3191 - accuracy:
0.5375 - val_loss: 1.1886 - val_accuracy: 0.5719
Epoch 187/200
80/80 [==============================] - 2s 22ms/step - loss: 1.3085 - accuracy:
0.5496 - val_loss: 1.2148 - val_accuracy: 0.5500
Epoch 188/200
80/80 [==============================] - 2s 23ms/step - loss: 1.3000 - accuracy:
0.5586 - val_loss: 1.1980 - val_accuracy: 0.5766
Epoch 189/200
80/80 [==============================] - 2s 22ms/step - loss: 1.3203 - accuracy:
0.5445 - val_loss: 1.2806 - val_accuracy: 0.5375
Epoch 190/200
80/80 [==============================] - 2s 23ms/step - loss: 1.2907 - accuracy:
0.5508 - val_loss: 1.2971 - val_accuracy: 0.5578
Epoch 191/200
80/80 [==============================] - 2s 22ms/step - loss: 1.3428 - accuracy:
0.5332 - val_loss: 1.2606 - val_accuracy: 0.5297
Epoch 192/200
80/80 [==============================] - 2s 25ms/step - loss: 1.3113 - accuracy:
0.5562 - val_loss: 1.2595 - val_accuracy: 0.5391
Epoch 193/200
80/80 [==============================] - 2s 23ms/step - loss: 1.3298 - accuracy:
0.5426 - val_loss: 1.2980 - val_accuracy: 0.5437
Epoch 194/200
80/80 [==============================] - 2s 22ms/step - loss: 1.3008 - accuracy:
0.5484 - val_loss: 1.2219 - val_accuracy: 0.5656
Epoch 195/200
```

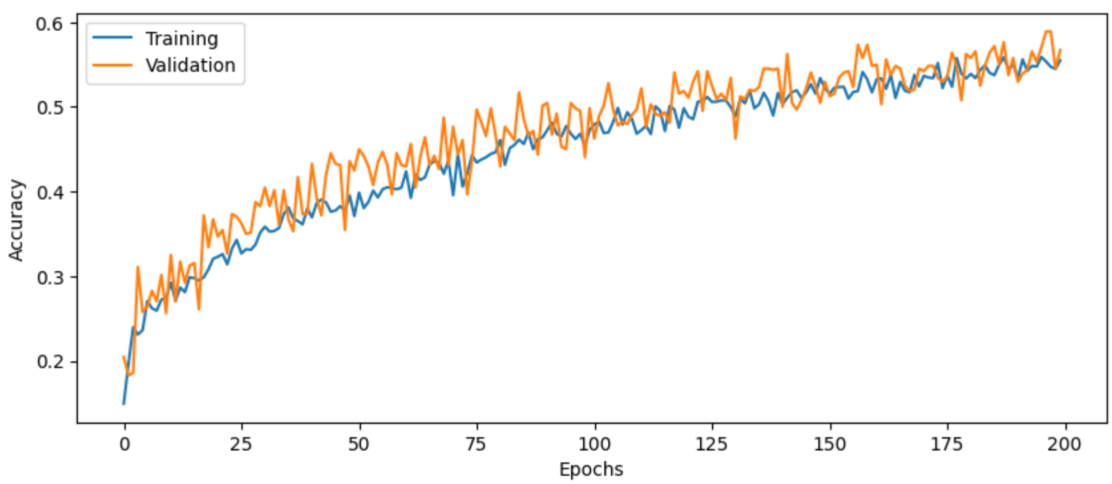
```
80/80 [==============================] - 2s 23ms/step - loss: 1.3260 - accuracy:
0.5477 - val_loss: 1.2233 - val_accuracy: 0.5531
Epoch 196/200
80/80 [==============================] - 2s 23ms/step - loss: 1.2951 - accuracy:
0.5590 - val_loss: 1.2122 - val_accuracy: 0.5703
Epoch 197/200
80/80 [==============================] - 2s 23ms/step - loss: 1.2868 - accuracy:
0.5535 - val_loss: 1.1900 - val_accuracy: 0.5891
Epoch 198/200
80/80 [==============================] - 2s 23ms/step - loss: 1.3030 - accuracy:
0.5473 - val_loss: 1.1405 - val_accuracy: 0.5891
Epoch 199/200
80/80 [==============================] - 2s 23ms/step - loss: 1.3236 - accuracy:
0.5449 - val_loss: 1.2590 - val_accuracy: 0.5453
Epoch 200/200
80/80 [==============================] - 2s 22ms/step - loss: 1.2955 - accuracy:
0.5551 - val_loss: 1.1543 - val_accuracy: 0.5672
```

[35]:
```
# Check if there is still a big difference in accuracy for original and rotated␣
↪test images

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

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

```
Test loss: 1.2567
Test accuracy: 0.5635
Test loss: 2.3526
Test accuracy: 0.2555
```

[36]:
```
# Plot the history from the training run
plot_results(history6)
```





Answer: Question 23

- The training accuracy increases at a slower rate per epoch when we train with augmentation. This is because the model have a harder time to learn the training data, since we have a lot more data. The data is also "harder" since they are augmented. We can increase the number of epochs to gain more training or increase the number of augmented images.

Question 24

- We can use different quality of images (blurriness) and deformation/distortion of images.

1.25 Part 20: Plot misclassified images

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

[37]:
```
# 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 5ms/step
```

[38]:
```
# Plot a few of them
plt.figure(figsize=(15,4))
perm = np.random.permutation(miss)
for i in range(18):
    im = (Xtest[perm[i]] + 1) * 127.5
    im = im.astype('int')
    label_correct = y_correct[perm[i]]
    label_pred = y_pred[perm[i]]

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

1.26 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: Question 25

- If we present images of other shapes, we will have different input shape. The first layer requires the correct input shape, so the model knows how many parameters it needs to learn.

Question 26

- We can apply the CNN to images of different sized if we reshape the images to 32 x 32.

1.27 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

[39]:
```
# Your code for using pre-trained ResNet 50 on 5 color images of your choice.
# The preprocessing should transform the image to a size that is expected by␣
↪the CNN.

from tensorflow.keras.applications.resnet50 import ResNet50, preprocess_input,␣
↪decode_predictions
from tensorflow.keras.preprocessing import image

model = ResNet50(weights='imagenet')
```

```python
path = ['ant.jpg', 'avocado.JPG', 'copybara.jpg', 'kangaroo.jpg', 'pig.jpg']
for index in range(0, 5):
    cur_image = image.load_img(path[index], target_size = (224, 224))

    #
    x = image.img_to_array(cur_image)
    x = np.expand_dims(x, axis=0)
    x = preprocess_input(x)

    predict = model.predict(x)

    print(f'The prediction is "{decode_predictions(predict)[0][0][1]}" for the␣
 ↪image: "{path[index][:-4]}"')

# Code used to answer the questions
# print(model.summary())
```

```
1/1 [==============================] - 1s 727ms/step
The prediction is "walking_stick" for the image: "ant"
1/1 [==============================] - 0s 82ms/step
The prediction is "buckeye" for the image: "avocado"
1/1 [==============================] - 0s 80ms/step
The prediction is "beaver" for the image: "copybara"
1/1 [==============================] - 0s 72ms/step
The prediction is "Arabian_camel" for the image: "kangaroo"
1/1 [==============================] - 0s 79ms/step
The prediction is "hog" for the image: "pig"
```

Answer: **Question 27**

- 48 convolutional layers.

Question 28

- 25,583,592 parameters

Question 29

- 224 x 224 images with 3 channels.

Question 30

- If this network would use the second derivative, the hessian matrix would be of size 25,583,592 x 25,583,592 which will be very computational heavy.