CNN_Lab_2024

May 8, 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).

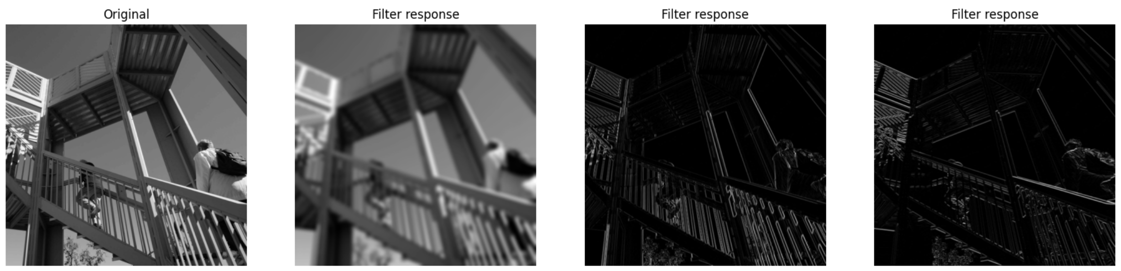
[ ]:
```
from scipy import signal
import numpy as np

# Get a test image

# Add datasets since scipy.misc.ascent deprecated since version 1.10.0
# and replaced by scipy.datasets.ascent
#from scipy import misc,datasets
#image = misc.ascent()

from scipy import datasets
image = datasets.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]])
```

```
Downloading file 'ascent.dat' from
'https://raw.githubusercontent.com/scipy/dataset-ascent/main/ascent.dat' to
'/root/.cache/scipy-data'.
```

[ ]:
```
filterResponseGauss = signal.convolve2d(image, gaussFilter, boundary='symm',␣
  ↪mode='same')
filterResponseSobelX = signal.convolve2d(image, sobelX, boundary='symm',␣
  ↪mode='same')
filterResponseSobelY = signal.convolve2d(image, sobelY, boundary='symm',␣
  ↪mode='same')
```

[ ]:
```
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?

```
[ ]: # Your code for checking sizes of image and filter responses
print("The size of image is:",image.shape)
print("The size of gaussFilter is:",gaussFilter.shape)
print("The size of sobelX is:",sobelX.shape)
print("The size of sobelY is:",sobelY.shape)

print("The size of filterResponseGauss(Same) is:",filterResponseGauss.shape)
print("The size of filterResponseSobelX(Same) is:",filterResponseSobelX.shape)
print("The size of filterResponseSobelY(Same) is:",filterResponseSobelY.shape)

filterResponseGaussWithValid = signal.convolve2d(image, gaussFilter,␣
 ↪boundary='symm', mode='valid')
filterResponseSobelXWithValid = signal.convolve2d(image, sobelX,␣
 ↪boundary='symm', mode='valid')
filterResponseSobelYWithValid = signal.convolve2d(image, sobelY,␣
 ↪boundary='symm', mode='valid')
```

```
print("The size of filterResponseGauss(Valid) is:",filterResponseGaussWithValid.
  ↪shape)
print("The size of filterResponseSobelX(Valid) is:
  ↪",filterResponseSobelXWithValid.shape)
print("The size of filterResponseSobelY(Valid) is:
  ↪",filterResponseSobelYWithValid.shape)
```

```
The size of image is: (512, 512)
The size of gaussFilter is: (15, 15)
The size of sobelX is: (3, 3)
The size of sobelY is: (3, 3)
The size of filterResponseGauss(Same) is: (512, 512)
The size of filterResponseSobelX(Same) is: (512, 512)
The size of filterResponseSobelY(Same) is: (512, 512)
The size of filterResponseGauss(Valid) is: (498, 498)
The size of filterResponseSobelX(Valid) is: (510, 510)
The size of filterResponseSobelY(Valid) is: (510, 510)
```

1.2.1 Answer to question 1

Gaussian filter will help to smooth the data and remove the noises.

Sobel X filter finds the first-order derivative in the X-direction. It detect the change in the X direction

Sobel Y filter finds the first-order derivative in the Y-direction. It detect the change in the Y direction

1.2.2 Answer to question 2

The CIFAR-10 dataset contains 60,000 32x32 color images in 10 different classes

For a single image in the CIFAR-10 dataset, it is a normal image, which contains RGB 3 channels. For a common color image, most of the cases, it contains 3 channels, which is RGB, but for some images, they will have another alpha channel which will become 4 channels for one image.For a grey image, it only contain one chanel which value is between 0 to 255.

1.2.3 Answer to question 3

The size of gaussFilter is: (15, 15)

The size of sobelX is: (3, 3)

The size of sobelY is: (3, 3)

1.2.4 Answer to question 4

With Same parameters:

The size of filterResponseGauss() is: (512, 512)

The size of filterResponseSobelX is: (512, 512)

The size of filterResponseSobelY is: (512, 512)

1.2.5 Answer to question 5

With Valid parameters:

The size of filterResponseGauss() is: (498, 498)

The size of filterResponseSobelX is: (510, 510)

The size of filterResponseSobelY is: (510, 510)

According to the document of signal.convolve2d, it has 3 mode values. Full,valid and same. Full is the default value. Function scipy.signal.convolve2d's first two parameters in1, in2 are image and filter matrix.

The following are from document of scipy.signal.convolve2d.

Full: The output is the full discrete linear convolution of the inputs(image).

Same: The output is the same size as in1, centered with respect to the full output.

Valid: The output consists only of those elements that do not rely on the zero-padding. In 'valid' mode, either in1 or in2 must be at least as large as the other in every dimension.

Since Original image's size is 512X512, GaussFilter's size is 15X15 and SobelX (SobelY)'s size is 3X3.

So for Valid case, (response dimension) = (image dimension) - (filter dimension) + 1 ,which is : 498 = 512 - 15 + 1 ; 510 = 512 - 3 + 1

1.2.6 Answer to question 6

Since size of image will get smaller after apply filter with valid parameter, so in CNNs with many layers, it is not suitable.

Fore example, a 512 X 512 image with 5 X 5 filter will shink to 0 X 0 after apply filter 128 times.

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.

[ ]:
```
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)
```

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.

1.4.1 Answer to question 7

For a colour image, it has R G B 3 channels, that's the reason why we need 7 x 7 x 3 filter instead of 7 x 7 filter(which is more suitable for a grey image)

1.4.2 Answer to question 8

The Conv2D layer in deep learning frameworks is designed for CNN and also optimized to run on GPU. On the other hand, the signal.convolve2d in SciPy perform more general 2D convolution operation. What Conv2D layer did is a standard 2D convolution operation.

Conv2D applies a 2D convolution between input image and kernels, and those kernels will change during the process of back propagation.(Grayscale image)

But in our case,we use 7 x 7 x 3 filter for color images, we can say that it is a 3D convolution between input image and kernels.

What it did are listed as follows:

1. Sliding a kernel across the input image. recalculate each point value with kernel.
2. Applying an activation function if needed.

1.4.3 Answer to question 9

We will use hign end RTX 4090 and Intel 14900 as example, every cuda core's speed is 2235 MHz and RTX 4090 has 16,000 CUDA cores. Intel 14900 ,on the other hand,has 8 performance-core with base frequency 3.2 Ghz and 16 efficient-cores with base frequency 2.4Ghz.

We can see that the base speed almost same and GPU has more computation units.

If we apply filters on 1000 image and 3 images at the same time.

In the case of 3 images case, if the image dimension is small, then the CPU wins, if the image size is relatively large, the CPU still win, since the base frequency of the CPU is higher than the GPU.

In the case of 1000 images case, in most cases, GPU win since GPU has more CUDA cores, and algorithms can utilize the parallel function of GPU.

But we also need to consider the time to copy from DDR to GDDR and copy back. Because copying lots of data from DDR to GDDR will take considerable time.

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.

```
[ ]: 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()

     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]

     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)))
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 [==============================] - 22s 0us/step
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.

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

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

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

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

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

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

     # Print the size of training data, validation data and test data
     print('Xtrain has Shape {}.'.format(Xtrain.shape))
     print('Ytrain has Shape {}.'.format(Ytrain.shape))

     print('Xval has Shape {}.'.format(Xval.shape))
     print('Yval has Shape {}.'.format(Yval.shape))

     print('Xtest has Shape {}.'.format(Xtest.shape))
     print('Ytest has Shape {}.'.format(Ytest.shape))
```

```
Xtrain has Shape (7500, 32, 32, 3).
Ytrain has Shape (7500, 1).
Xval has Shape (2500, 32, 32, 3).
Yval has Shape (2500, 1).
Xtest has Shape (2000, 32, 32, 3).
Ytest has Shape (2000, 1).
```

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.

```
[ ]: # 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

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

print("Before convertion \n")
# Print shapes before converting the labels
print('Ytrain has Shape {}.'.format(Ytrain.shape))
print('Yval has Shape {}.'.format(Yval.shape))
print('Ytest has Shape {}.'.format(Ytest.shape))

# Your code for converting Ytrain, Yval, Ytest to categorical
# we have 10 classes in the original dataset
Ytrainc = to_categorical(Ytrain, num_classes=10)
Yvalc = to_categorical(Yval, num_classes=10)
Ytestc = to_categorical(Ytest, num_classes=10)

print("\nAfter convertion \n")

# Print shapes after converting the labels
print('Ytrain has Shape {}.'.format(Ytrainc.shape))
print('Yval has Shape {}.'.format(Yval.shape))
print('Ytest has Shape {}.'.format(Ytest.shape))
```

```
Before convertion

Ytrain has Shape (7500, 1).
Yval has Shape (2500, 1).
Ytest has Shape (2000, 1).

After convertion

Ytrain has Shape (7500, 10).
Yval has Shape (2500, 1).
Ytest has Shape (2000, 1).
```

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

[ ]:
```
from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.layers import Input, Conv2D, BatchNormalization,␣
 ↪MaxPooling2D, Flatten, Dense, Dropout
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.layers import InputLayer
from tensorflow.keras import regularizers
from tensorflow.keras.losses import categorical_crossentropy

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

def build_CNN(input_shape, n_conv_layers=2, n_filters=16, n_dense_layers=0,␣
 ↪n_nodes=50, use_dropout=False, learning_rate=0.01,dropout_rate=0.5):

    # Setup a sequential model
    model = Sequential()
    # Add first convolutional layer to the model, requires input shape
    model.add(Conv2D(n_filters, kernel_size=(3, 3), padding='same',␣
↪activation="relu", input_shape=input_shape))

    model.add(BatchNormalization())
    model.add(MaxPooling2D(pool_size=(2, 2)))

    # Add remaining convolutional layers to the model, the number of filters␣
↪should increase a factor 2 for each layer
    double_filter = n_filters
    for i in range(n_conv_layers-1):
      double_filter = double_filter * 2
      model.add(Conv2D(double_filter, kernel_size=(3, 3),
                       padding='same',activation="relu"))
      model.add(BatchNormalization())
      model.add(MaxPooling2D(pool_size=(2, 2)))

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

    # Add intermediate dense layers
    for i in range(n_dense_layers):
      model.add(Dense(units=n_nodes, activation="relu"))
      if use_dropout == True:
            model.add(Dropout(dropout_rate))
      model.add(BatchNormalization())


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

    # Compile model
    opt = Adam(learning_rate=learning_rate)
    model.compile(loss= categorical_crossentropy, metrics=['accuracy'],␣
↪optimizer=opt)

    return model
```

[ ]:
```
# 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

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

n_conv_layers = 2

# Build model
model1 = build_CNN(input_shape = input_shape,n_conv_layers = n_conv_layers)

# Train the model  using training data and validation data
```

```
history1 = model1.fit(x=Xtrain,y=Ytrainc , epochs = 20 , validation_data =␣
  ↪(Xval,Yvalc), batch_size=batch_size)
```

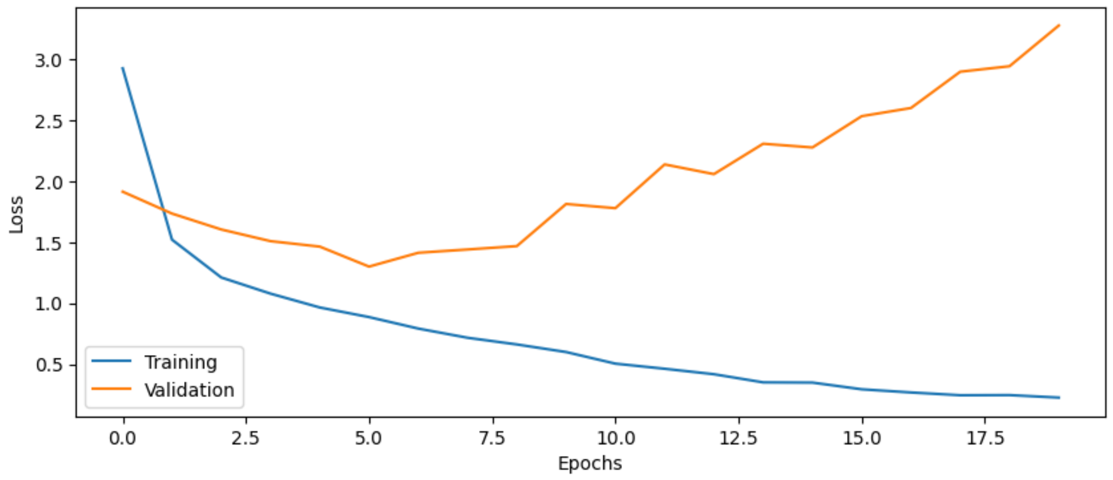
```
Epoch 1/20
75/75 [==============================] - 9s 33ms/step - loss: 2.9244 - accuracy:
0.3341 - val_loss: 1.9136 - val_accuracy: 0.3512
Epoch 2/20
75/75 [==============================] - 1s 16ms/step - loss: 1.5209 - accuracy:
0.5023 - val_loss: 1.7345 - val_accuracy: 0.3756
Epoch 3/20
75/75 [==============================] - 0s 5ms/step - loss: 1.2108 - accuracy:
0.5717 - val_loss: 1.6042 - val_accuracy: 0.4364
Epoch 4/20
75/75 [==============================] - 0s 5ms/step - loss: 1.0784 - accuracy:
0.6119 - val_loss: 1.5085 - val_accuracy: 0.4616
Epoch 5/20
75/75 [==============================] - 0s 5ms/step - loss: 0.9650 - accuracy:
0.6551 - val_loss: 1.4637 - val_accuracy: 0.4928
Epoch 6/20
75/75 [==============================] - 0s 5ms/step - loss: 0.8862 - accuracy:
0.6868 - val_loss: 1.2999 - val_accuracy: 0.5604
Epoch 7/20
75/75 [==============================] - 0s 5ms/step - loss: 0.7921 - accuracy:
0.7189 - val_loss: 1.4131 - val_accuracy: 0.5580
Epoch 8/20
75/75 [==============================] - 1s 7ms/step - loss: 0.7168 - accuracy:
0.7493 - val_loss: 1.4402 - val_accuracy: 0.5780
Epoch 9/20
75/75 [==============================] - 0s 7ms/step - loss: 0.6621 - accuracy:
0.7611 - val_loss: 1.4678 - val_accuracy: 0.5768
Epoch 10/20
75/75 [==============================] - 1s 11ms/step - loss: 0.5998 - accuracy:
0.7867 - val_loss: 1.8131 - val_accuracy: 0.5496
Epoch 11/20
75/75 [==============================] - 1s 8ms/step - loss: 0.5047 - accuracy:
0.8240 - val_loss: 1.7786 - val_accuracy: 0.5612
Epoch 12/20
75/75 [==============================] - 1s 18ms/step - loss: 0.4628 - accuracy:
0.8325 - val_loss: 2.1372 - val_accuracy: 0.5392
Epoch 13/20
75/75 [==============================] - 1s 15ms/step - loss: 0.4181 - accuracy:
0.8525 - val_loss: 2.0577 - val_accuracy: 0.5588
Epoch 14/20
75/75 [==============================] - 0s 5ms/step - loss: 0.3512 - accuracy:
0.8745 - val_loss: 2.3067 - val_accuracy: 0.5352
Epoch 15/20
75/75 [==============================] - 0s 5ms/step - loss: 0.3494 - accuracy:
```

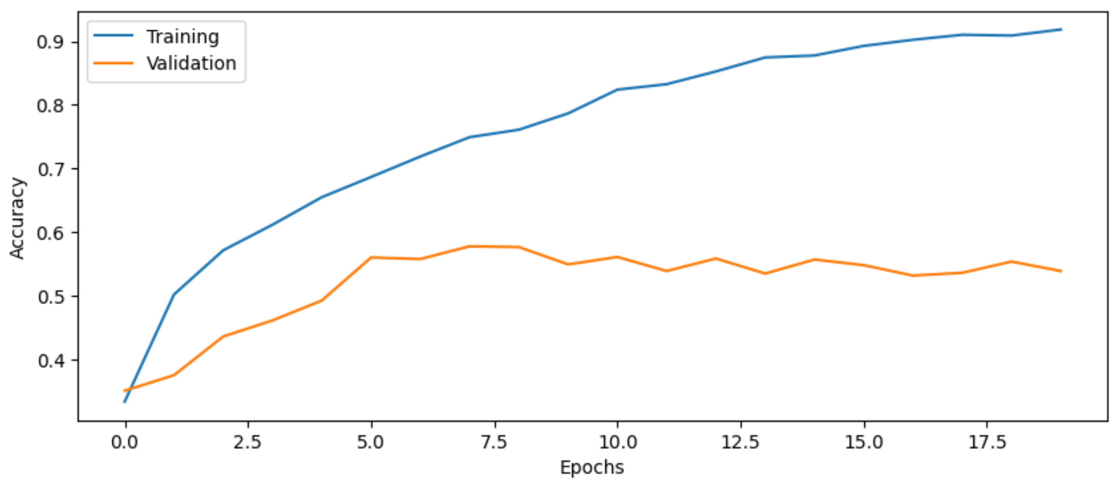
```
0.8776 - val_loss: 2.2760 - val_accuracy: 0.5572
Epoch 16/20
75/75 [==============================] - 1s 7ms/step - loss: 0.2945 - accuracy:
0.8927 - val_loss: 2.5324 - val_accuracy: 0.5484
Epoch 17/20
75/75 [==============================] - 0s 5ms/step - loss: 0.2685 - accuracy:
0.9023 - val_loss: 2.5995 - val_accuracy: 0.5320
Epoch 18/20
75/75 [==============================] - 0s 5ms/step - loss: 0.2458 - accuracy:
0.9101 - val_loss: 2.8968 - val_accuracy: 0.5364
Epoch 19/20
75/75 [==============================] - 1s 7ms/step - loss: 0.2470 - accuracy:
0.9089 - val_loss: 2.9418 - val_accuracy: 0.5540
Epoch 20/20
75/75 [==============================] - 0s 5ms/step - loss: 0.2265 - accuracy:
0.9184 - val_loss: 3.2754 - val_accuracy: 0.5392
```

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

```
63/63 [==============================] - 1s 15ms/step - loss: 3.1241 - accuracy:
0.5285
Test loss: 3.1241
Test accuracy: 0.5285
```

[ ]:
```
# 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?

1.14 Answer to question 10

According to the plot of history1 and the output of score, we found that after training 20 epochs, the training accuracy is 0.5392, meanwhile, test accuracy is 0.5285.

1.15 Answer to question 11

We need to set a smaller batch size in DNN because when we use a smaller batch size, backpropagation has more chances to update the parameters, making the model more stable and avoiding local min/max.

Another main reason is the dimension, input images have 32 x 32 x 3 dimensions compare to 92 in Lab1, which means we need more memory for each image, which limit the batch size.

1.16 2 convolutional layers, 1 intermediate dense layer (50 nodes)

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

n_conv_layers = 2
n_dense_layers = 1
```

```
n_nodes = 50

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

# Train the model  using training data and validation data
history2 = model2.fit(x=Xtrain,y=Ytrainc , epochs = 20 , validation_data =␣
 ↪(Xval,Yvalc), batch_size=batch_size)
```

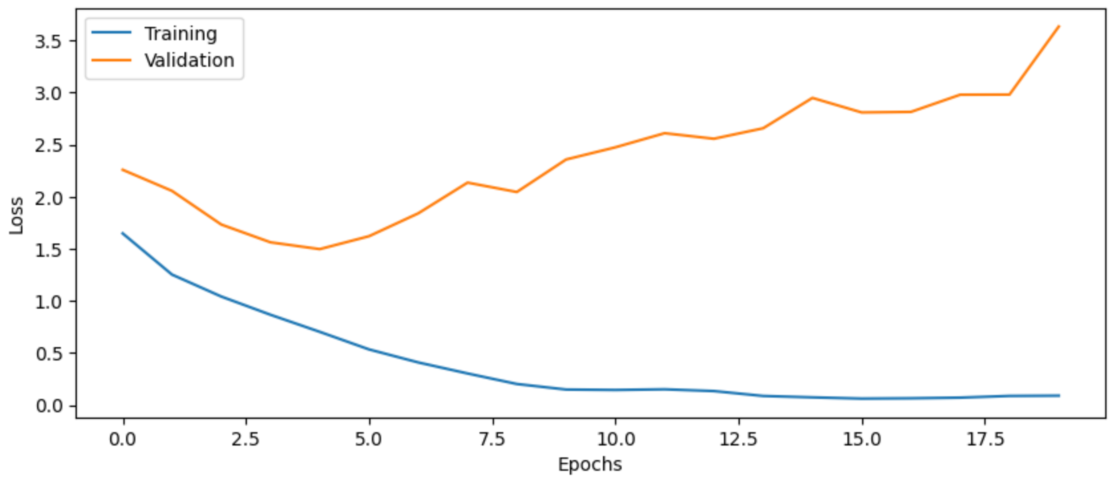
```
Epoch 1/20
75/75 [==============================] - 4s 14ms/step - loss: 1.6474 - accuracy:
0.4105 - val_loss: 2.2584 - val_accuracy: 0.2240
Epoch 2/20
75/75 [==============================] - 2s 24ms/step - loss: 1.2522 - accuracy:
0.5440 - val_loss: 2.0561 - val_accuracy: 0.2920
Epoch 3/20
75/75 [==============================] - 2s 24ms/step - loss: 1.0415 - accuracy:
0.6293 - val_loss: 1.7331 - val_accuracy: 0.4168
Epoch 4/20
75/75 [==============================] - 0s 6ms/step - loss: 0.8662 - accuracy:
0.6937 - val_loss: 1.5620 - val_accuracy: 0.4812
Epoch 5/20
75/75 [==============================] - 1s 8ms/step - loss: 0.7022 - accuracy:
0.7556 - val_loss: 1.4965 - val_accuracy: 0.5320
Epoch 6/20
75/75 [==============================] - 0s 6ms/step - loss: 0.5335 - accuracy:
0.8120 - val_loss: 1.6207 - val_accuracy: 0.5632
Epoch 7/20
75/75 [==============================] - 1s 10ms/step - loss: 0.4083 - accuracy:
0.8600 - val_loss: 1.8406 - val_accuracy: 0.5608
Epoch 8/20
75/75 [==============================] - 0s 6ms/step - loss: 0.3025 - accuracy:
0.8989 - val_loss: 2.1353 - val_accuracy: 0.5388
Epoch 9/20
75/75 [==============================] - 1s 9ms/step - loss: 0.2014 - accuracy:
0.9349 - val_loss: 2.0453 - val_accuracy: 0.5596
Epoch 10/20
75/75 [==============================] - 1s 10ms/step - loss: 0.1482 - accuracy:
0.9519 - val_loss: 2.3576 - val_accuracy: 0.5612
Epoch 11/20
75/75 [==============================] - 0s 6ms/step - loss: 0.1441 - accuracy:
0.9503 - val_loss: 2.4744 - val_accuracy: 0.5564
Epoch 12/20
75/75 [==============================] - 0s 6ms/step - loss: 0.1503 - accuracy:
```

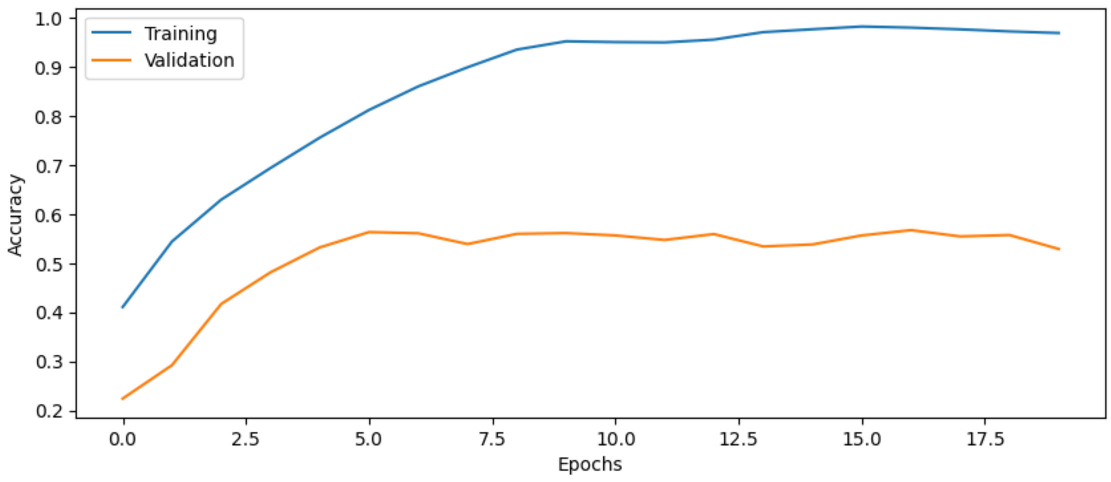
```
0.9496 - val_loss: 2.6100 - val_accuracy: 0.5472
Epoch 13/20
75/75 [==============================] - 2s 21ms/step - loss: 0.1334 - accuracy:
0.9553 - val_loss: 2.5567 - val_accuracy: 0.5592
Epoch 14/20
75/75 [==============================] - 1s 9ms/step - loss: 0.0864 - accuracy:
0.9705 - val_loss: 2.6574 - val_accuracy: 0.5340
Epoch 15/20
75/75 [==============================] - 1s 9ms/step - loss: 0.0731 - accuracy:
0.9764 - val_loss: 2.9489 - val_accuracy: 0.5380
Epoch 16/20
75/75 [==============================] - 0s 6ms/step - loss: 0.0612 - accuracy:
0.9820 - val_loss: 2.8092 - val_accuracy: 0.5564
Epoch 17/20
75/75 [==============================] - 1s 15ms/step - loss: 0.0641 - accuracy:
0.9797 - val_loss: 2.8145 - val_accuracy: 0.5672
Epoch 18/20
75/75 [==============================] - 0s 6ms/step - loss: 0.0704 - accuracy:
0.9763 - val_loss: 2.9797 - val_accuracy: 0.5544
Epoch 19/20
75/75 [==============================] - 0s 6ms/step - loss: 0.0868 - accuracy:
0.9720 - val_loss: 2.9810 - val_accuracy: 0.5572
Epoch 20/20
75/75 [==============================] - 1s 17ms/step - loss: 0.0893 - accuracy:
0.9688 - val_loss: 3.6345 - val_accuracy: 0.5288
```

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

```
63/63 [==============================] - 1s 11ms/step - loss: 3.8147 - accuracy:
0.5325
Test loss: 3.8147
Test accuracy: 0.5325
```

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





1.17 4 convolutional layers, 1 intermediate dense layer (50 nodes)

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

n_conv_layers = 4
n_dense_layers = 1
n_nodes = 50

# Build model
model3 = build_CNN(input_shape = input_shape,
                   n_conv_layers = n_conv_layers,
                   n_dense_layers = n_dense_layers,
                   n_nodes=n_nodes)

# Train the model  using training data and validation data
history3 = model3.fit(x=Xtrain,y=Ytrainc , epochs = 20 , validation_data =␣
 ↪(Xval,Yvalc), batch_size=batch_size)
```

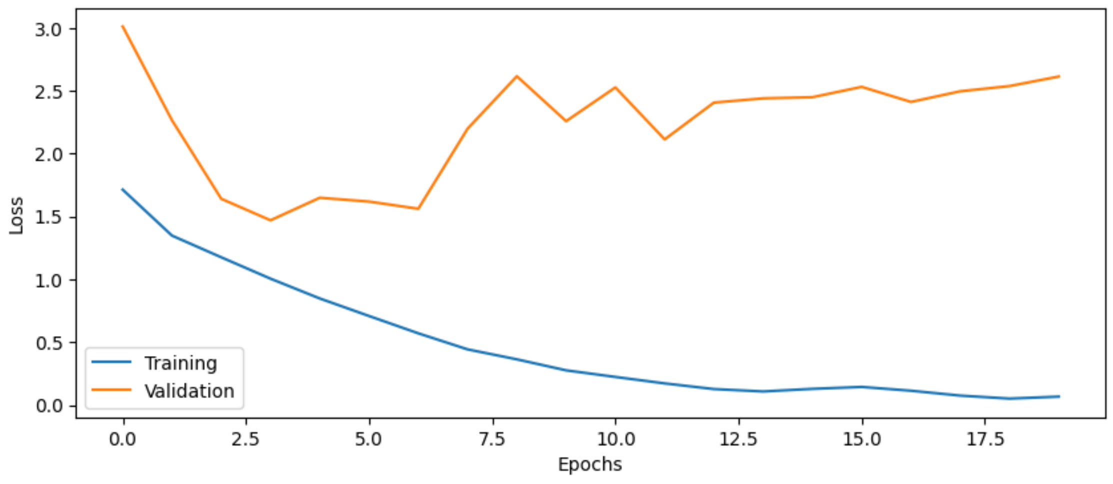
```
Epoch 1/20
75/75 [==============================] - 6s 12ms/step - loss: 1.7136 - accuracy:
0.3736 - val_loss: 3.0112 - val_accuracy: 0.2068
Epoch 2/20
75/75 [==============================] - 1s 10ms/step - loss: 1.3481 - accuracy:
0.5051 - val_loss: 2.2653 - val_accuracy: 0.2604
Epoch 3/20
75/75 [==============================] - 2s 22ms/step - loss: 1.1757 - accuracy:
0.5800 - val_loss: 1.6409 - val_accuracy: 0.4712
Epoch 4/20
75/75 [==============================] - 1s 8ms/step - loss: 1.0056 - accuracy:
0.6371 - val_loss: 1.4701 - val_accuracy: 0.5204
Epoch 5/20
75/75 [==============================] - 2s 21ms/step - loss: 0.8481 - accuracy:
0.6991 - val_loss: 1.6493 - val_accuracy: 0.5296
Epoch 6/20
75/75 [==============================] - 1s 18ms/step - loss: 0.7087 - accuracy:
0.7511 - val_loss: 1.6188 - val_accuracy: 0.5532
Epoch 7/20
75/75 [==============================] - 2s 28ms/step - loss: 0.5709 - accuracy:
0.7953 - val_loss: 1.5618 - val_accuracy: 0.5872
Epoch 8/20
75/75 [==============================] - 1s 7ms/step - loss: 0.4432 - accuracy:
0.8485 - val_loss: 2.1985 - val_accuracy: 0.5532
Epoch 9/20
75/75 [==============================] - 1s 19ms/step - loss: 0.3640 - accuracy:
0.8673 - val_loss: 2.6158 - val_accuracy: 0.5312
Epoch 10/20
75/75 [==============================] - 1s 7ms/step - loss: 0.2764 - accuracy:
0.9039 - val_loss: 2.2584 - val_accuracy: 0.5548
Epoch 11/20
75/75 [==============================] - 1s 19ms/step - loss: 0.2240 - accuracy:
0.9223 - val_loss: 2.5268 - val_accuracy: 0.5448
Epoch 12/20
75/75 [==============================] - 1s 7ms/step - loss: 0.1727 - accuracy:
0.9387 - val_loss: 2.1136 - val_accuracy: 0.5876
Epoch 13/20
75/75 [==============================] - 1s 20ms/step - loss: 0.1275 - accuracy:
0.9571 - val_loss: 2.4061 - val_accuracy: 0.5792
```

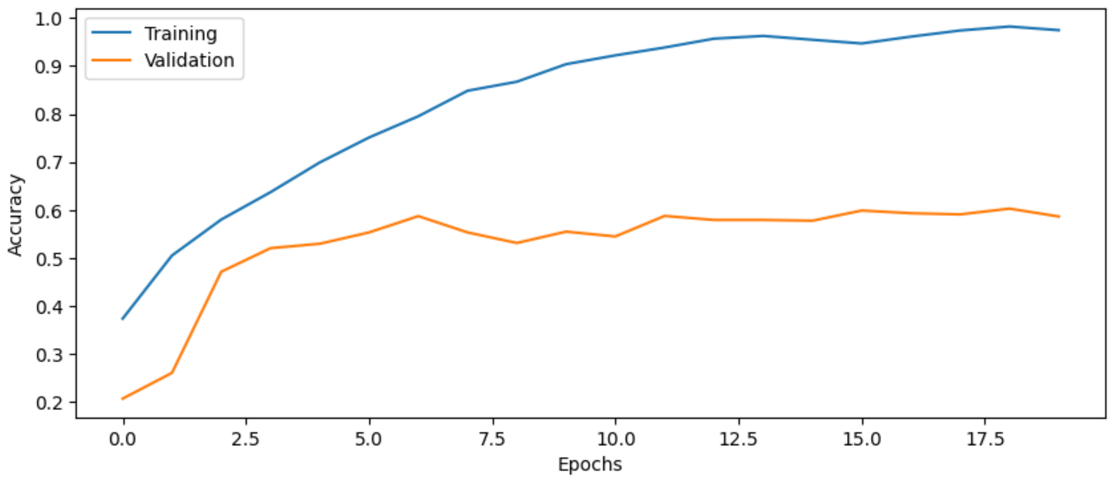
```
Epoch 14/20
75/75 [==============================] - 2s 27ms/step - loss: 0.1085 - accuracy:
0.9628 - val_loss: 2.4398 - val_accuracy: 0.5792
Epoch 15/20
75/75 [==============================] - 3s 38ms/step - loss: 0.1292 - accuracy:
0.9548 - val_loss: 2.4490 - val_accuracy: 0.5776
Epoch 16/20
75/75 [==============================] - 2s 28ms/step - loss: 0.1445 - accuracy:
0.9471 - val_loss: 2.5313 - val_accuracy: 0.5988
Epoch 17/20
75/75 [==============================] - 1s 7ms/step - loss: 0.1143 - accuracy:
0.9613 - val_loss: 2.4115 - val_accuracy: 0.5932
Epoch 18/20
75/75 [==============================] - 1s 18ms/step - loss: 0.0756 - accuracy:
0.9741 - val_loss: 2.4961 - val_accuracy: 0.5908
Epoch 19/20
75/75 [==============================] - 1s 7ms/step - loss: 0.0518 - accuracy:
0.9824 - val_loss: 2.5377 - val_accuracy: 0.6028
Epoch 20/20
75/75 [==============================] - 1s 8ms/step - loss: 0.0674 - accuracy:
0.9748 - val_loss: 2.6134 - val_accuracy: 0.5864
```

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

```
63/63 [==============================] - 1s 4ms/step - loss: 2.5770 - accuracy:
0.6100
Test loss: 2.5770
Test accuracy: 0.6100
```

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





1.18 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?

[ ]:
```
# Print network architecture
model3.summary()
```

```
Model: "sequential_2"

_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_4 (Conv2D)           (None, 32, 32, 16)        448

 batch_normalization_5 (Bat  (None, 32, 32, 16)        64
 chNormalization)

 max_pooling2d_4 (MaxPoolin  (None, 16, 16, 16)        0
 g2D)

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

 batch_normalization_6 (Bat  (None, 16, 16, 32)        128
 chNormalization)

 max_pooling2d_5 (MaxPoolin  (None, 8, 8, 32)          0
 g2D)

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

 batch_normalization_7 (Bat  (None, 8, 8, 64)          256
 chNormalization)

 max_pooling2d_6 (MaxPoolin  (None, 4, 4, 64)          0
 g2D)

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

 batch_normalization_8 (Bat  (None, 4, 4, 128)         512
 chNormalization)

 max_pooling2d_7 (MaxPoolin  (None, 2, 2, 128)         0
 g2D)

 flatten_2 (Flatten)         (None, 512)               0

 dense_3 (Dense)             (None, 50)                25650
 batch_normalization_9 (Bat  (None, 50)                200
 chNormalization)

 dense_4 (Dense)             (None, 10)                510

=================================================================
Total params: 124760 (487.34 KB)
Trainable params: 124180 (485.08 KB)
Non-trainable params: 580 (2.27 KB)
_________________________________________________________________
```

1.19 Answer to question 12

According to the summary of the model, we can find thet Trainable params number is 124180.

conv2d_7 contains most of the parameters which is 73856.

1.20 Answer to question 13

The input parameters of tf.keras.layers.Conv2D are filters, kernel_size, strides, padding, data_format, dilation_rate, groups, activation, use_bias, kernel_initializer, bias_initializer, kernel_regularizer, bias_regularizer, activity_regularizer, kernel_constraint, bias_constraint.

For more detailed info please check https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2D.

For the input of Conv2D, it is typically a 4D tensor representing a batch of input images tensor, the shape of this tensor is (batch_size, height, width, channels).In our case, the shape of the input tensor would be (100, 32, 32, 3)

Since we use the padding='same' when adding the Conv2D layer, the input and output dimensions do not change. In the case of the first Conv2D layer, channel number 3 becomes nfilter number 16. The remaining Conv2D's input and output remain the same.

For example, conv2d_4 (Conv2D):(32, 32, 16) and conv2d_5 (Conv2D):(32, 32, 16) at the very begining of this model.

1.21 Answer to question 14

Yes. The parameter data_format parameter in Conv2D can be set to "channels_last" or "channels_first".

"channels_last" corresponds to inputs with shape (batch_size, height, width, channels) while "channels_first" corresponds to inputs with shape (batch_size, channels, height, width).

Which means both of them set batch_size as its first parameter.

1.22 Answer to question 15

As discussed in question 13, if a convolutional layer that contains 128 filters is applied to an input with 32 channels, the number of channels in the output will be the filter number, which is 128.

1.23 Answer to question 16

The correct number of parameters in each Conv2D layer has the following formula. K x K x N, where K x K is the filter size N is the number of input channels.

For example, 3 x 3 x 3 means 3 x 3 filter and 3 input channels(color image).

1.24 Answer to question 17

Max pooling is performed on the convolutional layers of a CNN. It works like a convolutional layer, instead of performing a matrix multiplication, max pooling takes the maximum value within the window.

This means the output of the max pooling layer will get smaller compared to the input of the max pooling layer, which will reduce the dimensions of the feature maps.

Also, it helps to reduce the complexity of the model, with fewer parameters, the model is easier to generalize and helps to reduce the chance of overfitting. The related computation cost will also go down.

1.25 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?

2 Answer to question 18

The test accuracy with dropout is 0.6060 while the test accuracy without dropout which is 0.6100

3 Answer to question 19

we can apply the following regularization to Conv2D: According to the document of tf.keras.regularizers, we can apply the following regularizers to Conv2D.

1. L1: A regularizer that applies a L1 regularization penalty.
2. L1L2: A regularizer that applies both L1 and L2 regularization penalties.
3. L2: A regularizer that applies a L2 regularization penalty.
4. OrthogonalRegularizer: Regularizer that encourages input vectors to be orthogonal to each other.

The way to add L2 regularization to Conv2D is as follows, we use an example to illustrate it.

```
Conv2D(64, (3, 3), activation='relu', kernel_regularizer=regularizers.l2(0.01), input_shape=(3
```

3.1 4 convolutional layers, 1 intermediate dense layer (50 nodes), dropout

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

     n_conv_layers = 4
     n_dense_layers = 1
     n_nodes = 50

     # Build model
     model4 = build_CNN(input_shape = input_shape,
                        n_conv_layers = n_conv_layers,
                        n_dense_layers = n_dense_layers,
                        n_nodes=n_nodes,
                        use_dropout=True)

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

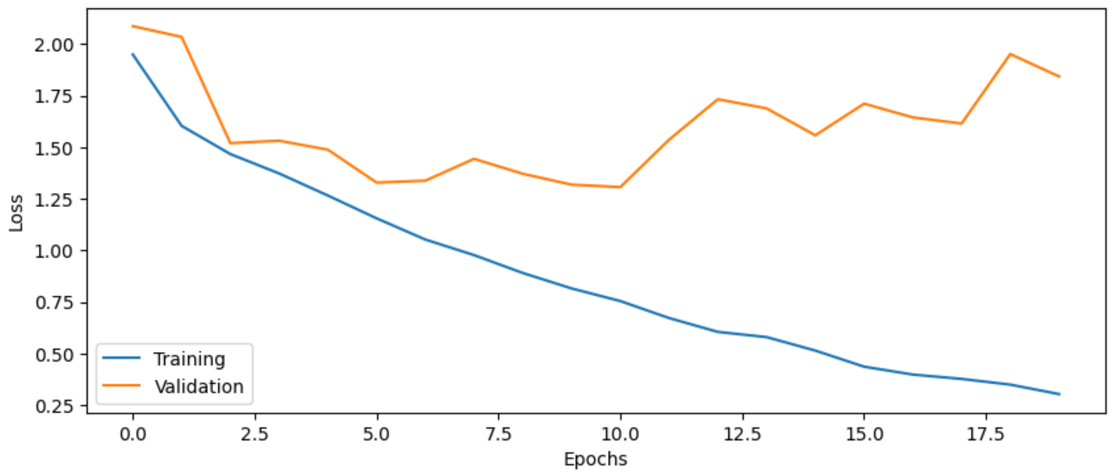
```
Epoch 1/20
75/75 [==============================] - 7s 29ms/step - loss: 1.9491 - accuracy:
0.3003 - val_loss: 2.0866 - val_accuracy: 0.2644
Epoch 2/20
75/75 [==============================] - 1s 10ms/step - loss: 1.6038 - accuracy:
0.3977 - val_loss: 2.0346 - val_accuracy: 0.2944
Epoch 3/20
75/75 [==============================] - 1s 11ms/step - loss: 1.4670 - accuracy:
0.4495 - val_loss: 1.5201 - val_accuracy: 0.4472
Epoch 4/20
75/75 [==============================] - 1s 11ms/step - loss: 1.3731 - accuracy:
0.4945 - val_loss: 1.5318 - val_accuracy: 0.4712
Epoch 5/20
75/75 [==============================] - 1s 8ms/step - loss: 1.2662 - accuracy:
0.5376 - val_loss: 1.4883 - val_accuracy: 0.4712
Epoch 6/20
75/75 [==============================] - 1s 19ms/step - loss: 1.1557 - accuracy:
0.5844 - val_loss: 1.3291 - val_accuracy: 0.5448
Epoch 7/20
75/75 [==============================] - 1s 10ms/step - loss: 1.0529 - accuracy:
0.6219 - val_loss: 1.3381 - val_accuracy: 0.5504
Epoch 8/20
75/75 [==============================] - 2s 22ms/step - loss: 0.9775 - accuracy:
0.6536 - val_loss: 1.4439 - val_accuracy: 0.5504
Epoch 9/20
75/75 [==============================] - 1s 9ms/step - loss: 0.8909 - accuracy:
```

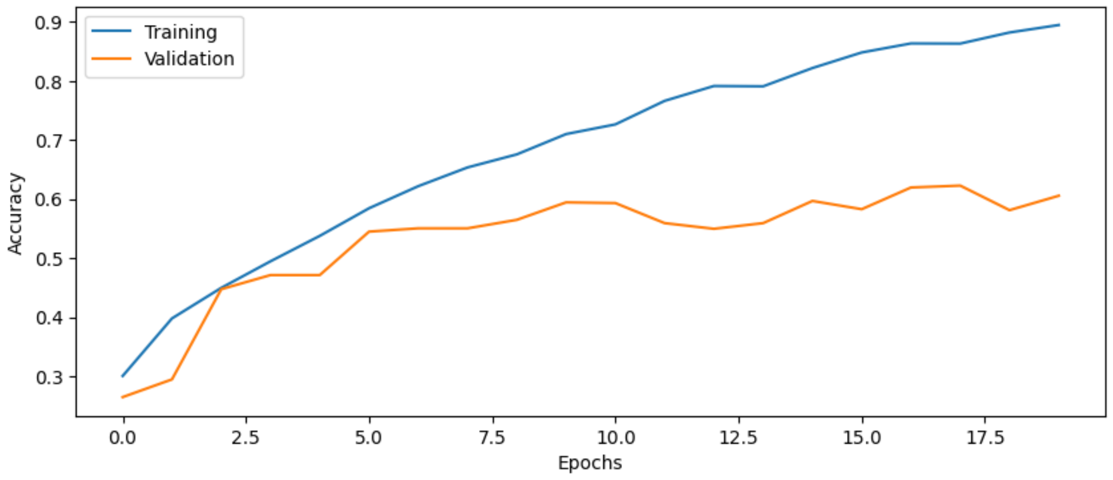
```
0.6757 - val_loss: 1.3716 - val_accuracy: 0.5648
Epoch 10/20
75/75 [==============================] - 1s 7ms/step - loss: 0.8161 - accuracy:
0.7101 - val_loss: 1.3189 - val_accuracy: 0.5944
Epoch 11/20
75/75 [==============================] - 1s 20ms/step - loss: 0.7550 - accuracy:
0.7265 - val_loss: 1.3070 - val_accuracy: 0.5932
Epoch 12/20
75/75 [==============================] - 1s 16ms/step - loss: 0.6728 - accuracy:
0.7664 - val_loss: 1.5367 - val_accuracy: 0.5592
Epoch 13/20
75/75 [==============================] - 1s 17ms/step - loss: 0.6057 - accuracy:
0.7916 - val_loss: 1.7326 - val_accuracy: 0.5496
Epoch 14/20
75/75 [==============================] - 1s 8ms/step - loss: 0.5804 - accuracy:
0.7911 - val_loss: 1.6884 - val_accuracy: 0.5592
Epoch 15/20
75/75 [==============================] - 1s 16ms/step - loss: 0.5151 - accuracy:
0.8220 - val_loss: 1.5579 - val_accuracy: 0.5968
Epoch 16/20
75/75 [==============================] - 1s 18ms/step - loss: 0.4372 - accuracy:
0.8484 - val_loss: 1.7112 - val_accuracy: 0.5828
Epoch 17/20
75/75 [==============================] - 1s 19ms/step - loss: 0.3989 - accuracy:
0.8637 - val_loss: 1.6446 - val_accuracy: 0.6196
Epoch 18/20
75/75 [==============================] - 1s 7ms/step - loss: 0.3779 - accuracy:
0.8635 - val_loss: 1.6150 - val_accuracy: 0.6228
Epoch 19/20
75/75 [==============================] - 1s 9ms/step - loss: 0.3502 - accuracy:
0.8823 - val_loss: 1.9514 - val_accuracy: 0.5812
Epoch 20/20
75/75 [==============================] - 1s 9ms/step - loss: 0.3044 - accuracy:
0.8949 - val_loss: 1.8430 - val_accuracy: 0.6056
```

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

```
63/63 [==============================] - 0s 3ms/step - loss: 1.7823 - accuracy:
0.6060
Test loss: 1.7823
Test accuracy: 0.6060
```

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





3.2 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?

3.3 Answer to question 20

The highest test accuracy we got is 0.6145

the best configuration are listed as follows.

```
batch_size = 200
epochs = 20
n_conv_layers = 4
n_filters = 16
n_dense_layers = 1
n_nodes = 100
learning_rate=0.01
use_dropout = False
```

3.4 Your best config

[ ]:
```
# Setup some training parameters
batch_size = 200
epochs = 20
input_shape = Xtrain.shape[1:]

n_conv_layers = 4
n_filters = 16
n_dense_layers = 1
n_nodes = 100
learning_rate=0.01

# Build model
model5 = build_CNN(input_shape = input_shape,
                   n_filters = n_filters,
                   n_conv_layers = n_conv_layers,
                   n_dense_layers = n_dense_layers,
                   n_nodes=n_nodes,
                   use_dropout=False,
                   learning_rate = learning_rate)

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

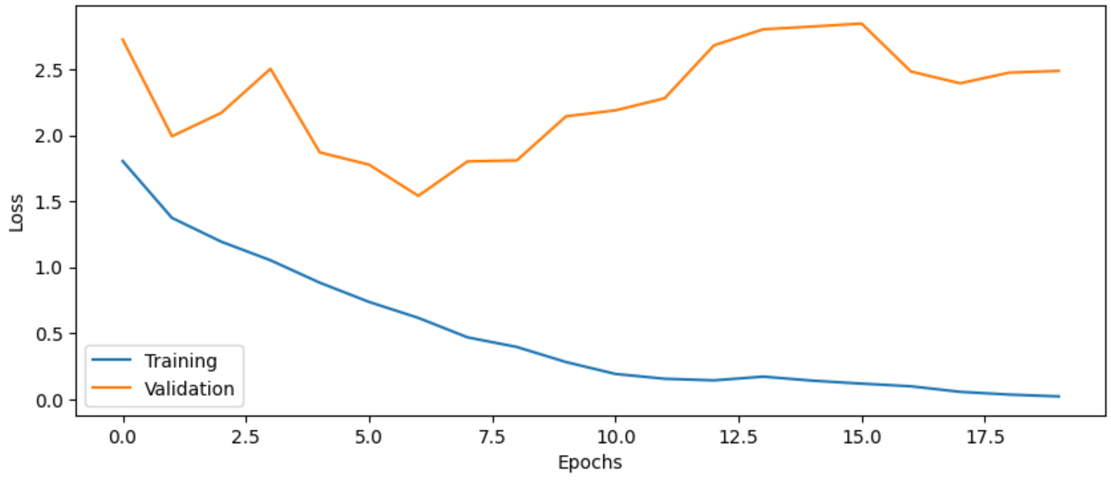
```
Epoch 1/20
38/38 [==============================] - 4s 21ms/step - loss: 1.8045 - accuracy:
0.3600 - val_loss: 2.7258 - val_accuracy: 0.2876
Epoch 2/20
38/38 [==============================] - 2s 56ms/step - loss: 1.3741 - accuracy:
0.4943 - val_loss: 1.9929 - val_accuracy: 0.3280
Epoch 3/20
38/38 [==============================] - 0s 12ms/step - loss: 1.1937 - accuracy:
0.5692 - val_loss: 2.1699 - val_accuracy: 0.3264
Epoch 4/20
38/38 [==============================] - 0s 11ms/step - loss: 1.0528 - accuracy:
0.6253 - val_loss: 2.5042 - val_accuracy: 0.2896
Epoch 5/20
```

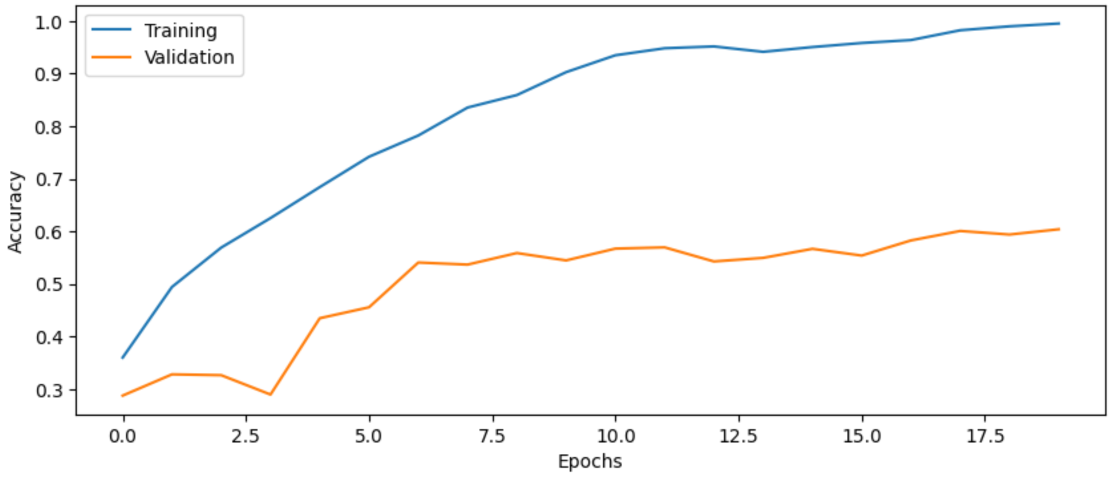
```
38/38 [==============================] - 0s 10ms/step - loss: 0.8827 - accuracy:
0.6843 - val_loss: 1.8706 - val_accuracy: 0.4348
Epoch 6/20
38/38 [==============================] - 0s 10ms/step - loss: 0.7369 - accuracy:
0.7420 - val_loss: 1.7779 - val_accuracy: 0.4556
Epoch 7/20
38/38 [==============================] - 1s 15ms/step - loss: 0.6159 - accuracy:
0.7824 - val_loss: 1.5412 - val_accuracy: 0.5408
Epoch 8/20
38/38 [==============================] - 0s 10ms/step - loss: 0.4684 - accuracy:
0.8356 - val_loss: 1.8024 - val_accuracy: 0.5368
Epoch 9/20
38/38 [==============================] - 1s 15ms/step - loss: 0.3954 - accuracy:
0.8593 - val_loss: 1.8098 - val_accuracy: 0.5588
Epoch 10/20
38/38 [==============================] - 0s 10ms/step - loss: 0.2809 - accuracy:
0.9029 - val_loss: 2.1442 - val_accuracy: 0.5448
Epoch 11/20
38/38 [==============================] - 0s 10ms/step - loss: 0.1910 - accuracy:
0.9351 - val_loss: 2.1893 - val_accuracy: 0.5672
Epoch 12/20
38/38 [==============================] - 1s 16ms/step - loss: 0.1544 - accuracy:
0.9484 - val_loss: 2.2808 - val_accuracy: 0.5696
Epoch 13/20
38/38 [==============================] - 0s 10ms/step - loss: 0.1422 - accuracy:
0.9519 - val_loss: 2.6818 - val_accuracy: 0.5428
Epoch 14/20
38/38 [==============================] - 1s 32ms/step - loss: 0.1705 - accuracy:
0.9416 - val_loss: 2.8040 - val_accuracy: 0.5496
Epoch 15/20
38/38 [==============================] - 1s 35ms/step - loss: 0.1400 - accuracy:
0.9507 - val_loss: 2.8248 - val_accuracy: 0.5668
Epoch 16/20
38/38 [==============================] - 1s 30ms/step - loss: 0.1175 - accuracy:
0.9584 - val_loss: 2.8473 - val_accuracy: 0.5540
Epoch 17/20
38/38 [==============================] - 1s 30ms/step - loss: 0.0973 - accuracy:
0.9640 - val_loss: 2.4836 - val_accuracy: 0.5828
Epoch 18/20
38/38 [==============================] - 1s 30ms/step - loss: 0.0552 - accuracy:
0.9827 - val_loss: 2.3943 - val_accuracy: 0.6008
Epoch 19/20
38/38 [==============================] - 1s 30ms/step - loss: 0.0344 - accuracy:
0.9901 - val_loss: 2.4757 - val_accuracy: 0.5940
Epoch 20/20
38/38 [==============================] - 0s 10ms/step - loss: 0.0205 - accuracy:
0.9955 - val_loss: 2.4886 - val_accuracy: 0.6040
```

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

```
63/63 [==============================] - 0s 3ms/step - loss: 2.4288 - accuracy:
0.6145
Test loss: 2.4288
Test accuracy: 0.6145
```

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





3.5 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.

3.6 Answer to question 21

The test accuracy for rotated test images is pretty low, which is around 0.2430 compared to the test accuracy for non-rotated test images is 0.6145

The reason behind this is the training data problem which causes the model generalization performance. Our training data does not contain rotated images, when we try to classify the rotated images, the model can not find a similar labelled image as a reference. So this one is the expected result.

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

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

    return images_rot
```

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

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

```
63/63 [==============================] - 1s 16ms/step - loss: 7.6294 - accuracy:
0.2430
Test loss: 7.6294
Test accuracy: 0.2430
```

3.7 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...

```
[ ]: # 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)
```

[ ]:
```
# 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

IDG = ImageDataGenerator(rotation_range=30,horizontal_flip=True,
                         vertical_flip=True,validation_split=0.2)

# Setup a flow for training data, assume that we can fit all images into CPU␣
↪memory
train_Flow = IDG.flow(x=Xtrain, y=Ytrain, batch_size=100, subset = "training")

# Setup a flow for validation data, assume that we can fit all images into CPU␣
↪memory
validate_Flow = IDG.flow(x=Xtrain, y=Ytrain, batch_size=100, subset =␣
↪"validation")
```

3.8 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?

3.9 Answer to question 22

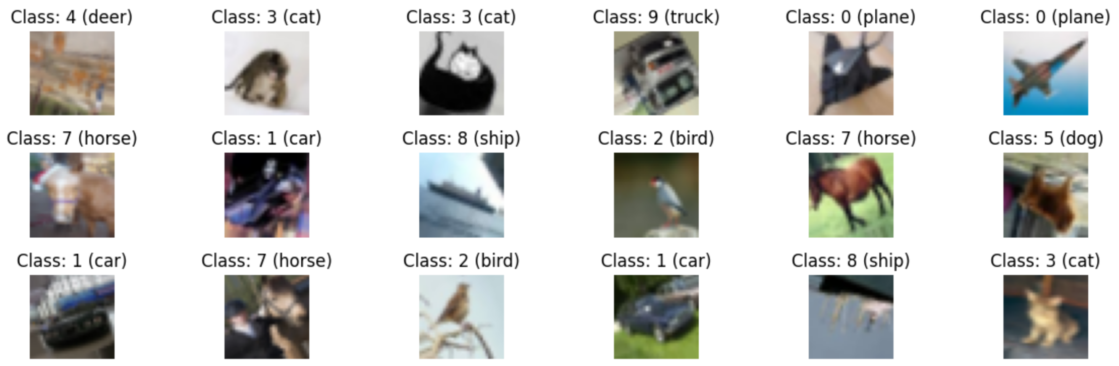
There are several ways to fit the data into memory.

Method 1: Decrease the training image size, but using this approach we will lose original features which will decrease the performance of the model.

Method 2: Minimize the batch size so that every batch can fit into the GPU memory. The disadvantage of this approach is the computational speed will be slower.

Method 3: If all the data can not fit into the main memory, we can stream the data from the hard drive or use virtual memory, both of them will decrease the computation performance and increase training time.

[ ]:
```
# Plot some augmented images
plot_datagen = IDG.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()
```



3.10 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?

3.11 Answer to question 23

As can be seen from the plot below, the training accuracy without augmentation increases much faster compared to the training accuracy with augmentation.

The reason behind this is augmentation create new images which means we showing new images in every epoch, instead of the same images over and over.

3.12 Answer to question 24

In the above code, we already use rotation, horizontal_flip, and vertical_flip, we can also apply Scaling, Brightness adjustment, Contrast adjustment, Noise addition, Color jitter, Blur, Elastic deformation etc to the original image to create "new" training data.

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

n_conv_layers = 4
n_dense_layers = 1
n_nodes = 100
learning_rate=0.01

# Build model (your best config)
model6 = build_CNN(input_shape = input_shape,
                   n_conv_layers = n_conv_layers,
                   n_dense_layers = n_dense_layers,
                   n_nodes=n_nodes,
                   use_dropout=False,

                   learning_rate = learning_rate)

validation_split=0.2


history6 = model6.fit(train_Flow,
                      validation_data = validate_Flow,
                      steps_per_epoch = len(Xtrain)*(1 - validation_split)/
 ↪batch_size,
                      validation_steps = len(Xtrain)*validation_split/
 ↪batch_size,
                      batch_size = batch_size,
                      epochs = epochs)
```

```
Epoch 1/200
80/80 [==============================] - 11s 100ms/step - loss: 1.9114 -
accuracy: 0.3108 - val_loss: 2.1167 - val_accuracy: 0.2835
Epoch 2/200
80/80 [==============================] - 5s 63ms/step - loss: 1.6509 - accuracy:
0.3907 - val_loss: 2.1387 - val_accuracy: 0.2620
Epoch 3/200
80/80 [==============================] - 7s 90ms/step - loss: 1.5576 - accuracy:
0.4274 - val_loss: 1.5858 - val_accuracy: 0.4255
Epoch 4/200
80/80 [==============================] - 6s 74ms/step - loss: 1.5035 - accuracy:
```

```
0.4526 - val_loss: 1.4995 - val_accuracy: 0.4620
Epoch 5/200
80/80 [==============================] - 6s 78ms/step - loss: 1.4145 - accuracy:
0.4794 - val_loss: 1.8231 - val_accuracy: 0.4090
Epoch 6/200
80/80 [==============================] - 7s 93ms/step - loss: 1.3790 - accuracy:
0.4958 - val_loss: 1.5926 - val_accuracy: 0.4395
Epoch 7/200
80/80 [==============================] - 8s 101ms/step - loss: 1.3268 -
accuracy: 0.5169 - val_loss: 1.4483 - val_accuracy: 0.5075
Epoch 8/200
80/80 [==============================] - 6s 81ms/step - loss: 1.2778 - accuracy:
0.5410 - val_loss: 1.5094 - val_accuracy: 0.4815
Epoch 9/200
80/80 [==============================] - 7s 83ms/step - loss: 1.2300 - accuracy:
0.5573 - val_loss: 1.3376 - val_accuracy: 0.5375
Epoch 10/200
80/80 [==============================] - 7s 75ms/step - loss: 1.2032 - accuracy:
0.5640 - val_loss: 1.4489 - val_accuracy: 0.5075
Epoch 11/200
80/80 [==============================] - 6s 75ms/step - loss: 1.1988 - accuracy:
0.5689 - val_loss: 1.5566 - val_accuracy: 0.5025
Epoch 12/200
80/80 [==============================] - 6s 71ms/step - loss: 1.1544 - accuracy:
0.5832 - val_loss: 1.4115 - val_accuracy: 0.5295
Epoch 13/200
80/80 [==============================] - 7s 87ms/step - loss: 1.1298 - accuracy:
0.5930 - val_loss: 1.3731 - val_accuracy: 0.5345
Epoch 14/200
80/80 [==============================] - 8s 96ms/step - loss: 1.1059 - accuracy:
0.6004 - val_loss: 1.1953 - val_accuracy: 0.5735
Epoch 15/200
80/80 [==============================] - 5s 69ms/step - loss: 1.1094 - accuracy:
0.6005 - val_loss: 1.3345 - val_accuracy: 0.5675
Epoch 16/200
80/80 [==============================] - 6s 75ms/step - loss: 1.0606 - accuracy:
0.6159 - val_loss: 1.3053 - val_accuracy: 0.5400
Epoch 17/200
80/80 [==============================] - 7s 86ms/step - loss: 1.0403 - accuracy:
0.6271 - val_loss: 1.4138 - val_accuracy: 0.5390
Epoch 18/200
80/80 [==============================] - 6s 71ms/step - loss: 1.0195 - accuracy:
0.6291 - val_loss: 1.3162 - val_accuracy: 0.5560
Epoch 19/200
80/80 [==============================] - 11s 140ms/step - loss: 1.0011 -
accuracy: 0.6434 - val_loss: 1.2662 - val_accuracy: 0.5815
Epoch 20/200
80/80 [==============================] - 6s 71ms/step - loss: 0.9958 - accuracy:
```

```
0.6450 - val_loss: 1.1891 - val_accuracy: 0.5855
Epoch 21/200
80/80 [==============================] - 6s 71ms/step - loss: 0.9645 - accuracy:
0.6522 - val_loss: 1.2403 - val_accuracy: 0.6010
Epoch 22/200
80/80 [==============================] - 7s 91ms/step - loss: 0.9581 - accuracy:
0.6622 - val_loss: 1.3503 - val_accuracy: 0.5505
Epoch 23/200
80/80 [==============================] - 7s 86ms/step - loss: 0.9502 - accuracy:
0.6644 - val_loss: 1.1879 - val_accuracy: 0.5860
Epoch 24/200
80/80 [==============================] - 7s 80ms/step - loss: 0.9285 - accuracy:
0.6664 - val_loss: 1.2090 - val_accuracy: 0.5955
Epoch 25/200
80/80 [==============================] - 6s 76ms/step - loss: 0.9181 - accuracy:
0.6693 - val_loss: 1.2675 - val_accuracy: 0.5810
Epoch 26/200
80/80 [==============================] - 7s 91ms/step - loss: 0.9072 - accuracy:
0.6794 - val_loss: 1.2332 - val_accuracy: 0.5945
Epoch 27/200
80/80 [==============================] - 6s 76ms/step - loss: 0.8971 - accuracy:
0.6770 - val_loss: 1.1943 - val_accuracy: 0.6120
Epoch 28/200
80/80 [==============================] - 8s 97ms/step - loss: 0.8901 - accuracy:
0.6791 - val_loss: 1.1651 - val_accuracy: 0.5970
Epoch 29/200
80/80 [==============================] - 8s 95ms/step - loss: 0.8800 - accuracy:
0.6837 - val_loss: 1.1885 - val_accuracy: 0.5880
Epoch 30/200
80/80 [==============================] - 6s 72ms/step - loss: 0.8449 - accuracy:
0.6974 - val_loss: 1.2169 - val_accuracy: 0.6145
Epoch 31/200
80/80 [==============================] - 6s 78ms/step - loss: 0.8531 - accuracy:
0.6902 - val_loss: 1.1959 - val_accuracy: 0.5960
Epoch 32/200
80/80 [==============================] - 7s 87ms/step - loss: 0.8227 - accuracy:
0.7070 - val_loss: 1.1890 - val_accuracy: 0.6010
Epoch 33/200
80/80 [==============================] - 6s 74ms/step - loss: 0.8178 - accuracy:
0.7032 - val_loss: 1.1371 - val_accuracy: 0.6135
Epoch 34/200
80/80 [==============================] - 6s 79ms/step - loss: 0.8190 - accuracy:
0.7082 - val_loss: 1.2281 - val_accuracy: 0.5940
Epoch 35/200
80/80 [==============================] - 6s 66ms/step - loss: 0.8187 - accuracy:
0.7074 - val_loss: 1.2244 - val_accuracy: 0.5915
Epoch 36/200
80/80 [==============================] - 6s 77ms/step - loss: 0.7940 - accuracy:
```

```
0.7157 - val_loss: 1.1852 - val_accuracy: 0.6205
Epoch 37/200
80/80 [==============================] - 6s 79ms/step - loss: 0.7970 - accuracy:
0.7144 - val_loss: 1.1963 - val_accuracy: 0.6025
Epoch 38/200
80/80 [==============================] - 6s 74ms/step - loss: 0.7856 - accuracy:
0.7216 - val_loss: 1.1698 - val_accuracy: 0.6135
Epoch 39/200
80/80 [==============================] - 6s 81ms/step - loss: 0.7639 - accuracy:
0.7234 - val_loss: 1.1607 - val_accuracy: 0.6115
Epoch 40/200
80/80 [==============================] - 6s 64ms/step - loss: 0.7683 - accuracy:
0.7228 - val_loss: 1.1744 - val_accuracy: 0.6210
Epoch 41/200
80/80 [==============================] - 8s 97ms/step - loss: 0.7503 - accuracy:
0.7341 - val_loss: 1.2665 - val_accuracy: 0.6010
Epoch 42/200
80/80 [==============================] - 6s 77ms/step - loss: 0.7465 - accuracy:
0.7334 - val_loss: 1.1982 - val_accuracy: 0.6265
Epoch 43/200
80/80 [==============================] - 7s 74ms/step - loss: 0.7353 - accuracy:
0.7330 - val_loss: 1.1814 - val_accuracy: 0.6230
Epoch 44/200
80/80 [==============================] - 7s 83ms/step - loss: 0.7329 - accuracy:
0.7411 - val_loss: 1.3201 - val_accuracy: 0.5955
Epoch 45/200
80/80 [==============================] - 7s 84ms/step - loss: 0.7492 - accuracy:
0.7280 - val_loss: 1.3142 - val_accuracy: 0.6010
Epoch 46/200
80/80 [==============================] - 6s 73ms/step - loss: 0.7093 - accuracy:
0.7504 - val_loss: 1.1705 - val_accuracy: 0.6315
Epoch 47/200
80/80 [==============================] - 6s 79ms/step - loss: 0.7119 - accuracy:
0.7431 - val_loss: 1.1823 - val_accuracy: 0.6175
Epoch 48/200
80/80 [==============================] - 7s 92ms/step - loss: 0.7043 - accuracy:
0.7458 - val_loss: 1.1771 - val_accuracy: 0.6270
Epoch 49/200
80/80 [==============================] - 6s 71ms/step - loss: 0.6822 - accuracy:
0.7567 - val_loss: 1.2071 - val_accuracy: 0.6325
Epoch 50/200
80/80 [==============================] - 6s 74ms/step - loss: 0.7068 - accuracy:
0.7485 - val_loss: 1.2544 - val_accuracy: 0.6115
Epoch 51/200
80/80 [==============================] - 8s 96ms/step - loss: 0.6822 - accuracy:
0.7586 - val_loss: 1.1437 - val_accuracy: 0.6325
Epoch 52/200
80/80 [==============================] - 6s 79ms/step - loss: 0.6906 - accuracy:
```

```
0.7520 - val_loss: 1.2260 - val_accuracy: 0.6225
Epoch 53/200
80/80 [==============================] - 5s 67ms/step - loss: 0.6794 - accuracy:
0.7559 - val_loss: 1.2260 - val_accuracy: 0.6190
Epoch 54/200
80/80 [==============================] - 7s 83ms/step - loss: 0.6590 - accuracy:
0.7639 - val_loss: 1.2509 - val_accuracy: 0.6025
Epoch 55/200
80/80 [==============================] - 6s 74ms/step - loss: 0.6382 - accuracy:
0.7731 - val_loss: 1.3300 - val_accuracy: 0.6130
Epoch 56/200
80/80 [==============================] - 6s 72ms/step - loss: 0.6160 - accuracy:
0.7820 - val_loss: 1.1546 - val_accuracy: 0.6410
Epoch 57/200
80/80 [==============================] - 7s 88ms/step - loss: 0.6382 - accuracy:
0.7740 - val_loss: 1.2589 - val_accuracy: 0.6285
Epoch 58/200
80/80 [==============================] - 8s 98ms/step - loss: 0.6193 - accuracy:
0.7763 - val_loss: 1.1713 - val_accuracy: 0.6375
Epoch 59/200
80/80 [==============================] - 6s 75ms/step - loss: 0.6412 - accuracy:
0.7739 - val_loss: 1.1871 - val_accuracy: 0.6305
Epoch 60/200
80/80 [==============================] - 8s 98ms/step - loss: 0.6188 - accuracy:
0.7843 - val_loss: 1.3135 - val_accuracy: 0.6150
Epoch 61/200
80/80 [==============================] - 7s 82ms/step - loss: 0.6018 - accuracy:
0.7857 - val_loss: 1.2016 - val_accuracy: 0.6300
Epoch 62/200
80/80 [==============================] - 6s 69ms/step - loss: 0.6052 - accuracy:
0.7809 - val_loss: 1.2636 - val_accuracy: 0.6235
Epoch 63/200
80/80 [==============================] - 6s 78ms/step - loss: 0.6024 - accuracy:
0.7875 - val_loss: 1.2463 - val_accuracy: 0.6345
Epoch 64/200
80/80 [==============================] - 6s 81ms/step - loss: 0.6204 - accuracy:
0.7786 - val_loss: 1.1396 - val_accuracy: 0.6465
Epoch 65/200
80/80 [==============================] - 6s 75ms/step - loss: 0.6248 - accuracy:
0.7794 - val_loss: 1.1702 - val_accuracy: 0.6355
Epoch 66/200
80/80 [==============================] - 7s 85ms/step - loss: 0.5713 - accuracy:
0.7910 - val_loss: 1.2411 - val_accuracy: 0.6265
Epoch 67/200
80/80 [==============================] - 6s 72ms/step - loss: 0.5724 - accuracy:
0.7934 - val_loss: 1.3059 - val_accuracy: 0.6155
Epoch 68/200
80/80 [==============================] - 6s 71ms/step - loss: 0.5802 - accuracy:
```

```
0.7915 - val_loss: 1.2110 - val_accuracy: 0.6265
Epoch 69/200
80/80 [==============================] - 5s 57ms/step - loss: 0.5731 - accuracy:
0.7972 - val_loss: 1.2315 - val_accuracy: 0.6320
Epoch 70/200
80/80 [==============================] - 6s 77ms/step - loss: 0.5767 - accuracy:
0.7941 - val_loss: 1.2156 - val_accuracy: 0.6345
Epoch 71/200
80/80 [==============================] - 7s 87ms/step - loss: 0.5465 - accuracy:
0.8075 - val_loss: 1.2595 - val_accuracy: 0.6390
Epoch 72/200
80/80 [==============================] - 7s 86ms/step - loss: 0.5660 - accuracy:
0.7974 - val_loss: 1.2373 - val_accuracy: 0.6350
Epoch 73/200
80/80 [==============================] - 6s 70ms/step - loss: 0.5692 - accuracy:
0.7972 - val_loss: 1.2501 - val_accuracy: 0.6380
Epoch 74/200
80/80 [==============================] - 8s 101ms/step - loss: 0.5457 -
accuracy: 0.8086 - val_loss: 1.3328 - val_accuracy: 0.6130
Epoch 75/200
80/80 [==============================] - 5s 65ms/step - loss: 0.5511 - accuracy:
0.8098 - val_loss: 1.2837 - val_accuracy: 0.6350
Epoch 76/200
80/80 [==============================] - 6s 70ms/step - loss: 0.5384 - accuracy:
0.8098 - val_loss: 1.2818 - val_accuracy: 0.6260
Epoch 77/200
80/80 [==============================] - 7s 92ms/step - loss: 0.5347 - accuracy:
0.8111 - val_loss: 1.2390 - val_accuracy: 0.6475
Epoch 78/200
80/80 [==============================] - 8s 99ms/step - loss: 0.5391 - accuracy:
0.8071 - val_loss: 1.2805 - val_accuracy: 0.6290
Epoch 79/200
80/80 [==============================] - 6s 73ms/step - loss: 0.5294 - accuracy:
0.8092 - val_loss: 1.2596 - val_accuracy: 0.6450
Epoch 80/200
80/80 [==============================] - 8s 97ms/step - loss: 0.5273 - accuracy:
0.8067 - val_loss: 1.2682 - val_accuracy: 0.6305
Epoch 81/200
80/80 [==============================] - 7s 85ms/step - loss: 0.5186 - accuracy:
0.8148 - val_loss: 1.3125 - val_accuracy: 0.6395
Epoch 82/200
80/80 [==============================] - 6s 77ms/step - loss: 0.5220 - accuracy:
0.8145 - val_loss: 1.3854 - val_accuracy: 0.6250
Epoch 83/200
80/80 [==============================] - 7s 84ms/step - loss: 0.5142 - accuracy:
0.8180 - val_loss: 1.3521 - val_accuracy: 0.6180
Epoch 84/200
80/80 [==============================] - 6s 81ms/step - loss: 0.5074 - accuracy:
```

```
0.8216 - val_loss: 1.2668 - val_accuracy: 0.6445
Epoch 85/200
80/80 [==============================] - 6s 79ms/step - loss: 0.5017 - accuracy:
0.8239 - val_loss: 1.2947 - val_accuracy: 0.6435
Epoch 86/200
80/80 [==============================] - 7s 84ms/step - loss: 0.5103 - accuracy:
0.8190 - val_loss: 1.2842 - val_accuracy: 0.6450
Epoch 87/200
80/80 [==============================] - 6s 72ms/step - loss: 0.5101 - accuracy:
0.8175 - val_loss: 1.3629 - val_accuracy: 0.6235
Epoch 88/200
80/80 [==============================] - 5s 65ms/step - loss: 0.4829 - accuracy:
0.8281 - val_loss: 1.2654 - val_accuracy: 0.6375
Epoch 89/200
80/80 [==============================] - 7s 92ms/step - loss: 0.4876 - accuracy:
0.8273 - val_loss: 1.4031 - val_accuracy: 0.6250
Epoch 90/200
80/80 [==============================] - 7s 85ms/step - loss: 0.4672 - accuracy:
0.8363 - val_loss: 1.3299 - val_accuracy: 0.6320
Epoch 91/200
80/80 [==============================] - 6s 74ms/step - loss: 0.4700 - accuracy:
0.8304 - val_loss: 1.3280 - val_accuracy: 0.6215
Epoch 92/200
80/80 [==============================] - 6s 80ms/step - loss: 0.4623 - accuracy:
0.8346 - val_loss: 1.3935 - val_accuracy: 0.6240
Epoch 93/200
80/80 [==============================] - 5s 65ms/step - loss: 0.4757 - accuracy:
0.8317 - val_loss: 1.3320 - val_accuracy: 0.6380
Epoch 94/200
80/80 [==============================] - 6s 75ms/step - loss: 0.4663 - accuracy:
0.8406 - val_loss: 1.2854 - val_accuracy: 0.6435
Epoch 95/200
80/80 [==============================] - 6s 75ms/step - loss: 0.4818 - accuracy:
0.8290 - val_loss: 1.3700 - val_accuracy: 0.6375
Epoch 96/200
80/80 [==============================] - 7s 83ms/step - loss: 0.4913 - accuracy:
0.8264 - val_loss: 1.2242 - val_accuracy: 0.6395
Epoch 97/200
80/80 [==============================] - 6s 75ms/step - loss: 0.4511 - accuracy:
0.8379 - val_loss: 1.3319 - val_accuracy: 0.6280
Epoch 98/200
80/80 [==============================] - 8s 98ms/step - loss: 0.4562 - accuracy:
0.8354 - val_loss: 1.3432 - val_accuracy: 0.6250
Epoch 99/200
80/80 [==============================] - 7s 89ms/step - loss: 0.4675 - accuracy:
0.8319 - val_loss: 1.3110 - val_accuracy: 0.6530
Epoch 100/200
80/80 [==============================] - 7s 90ms/step - loss: 0.4488 - accuracy:
```

```
0.8424 - val_loss: 1.3546 - val_accuracy: 0.6505
Epoch 101/200
80/80 [==============================] - 6s 75ms/step - loss: 0.4497 - accuracy:
0.8405 - val_loss: 1.3621 - val_accuracy: 0.6445
Epoch 102/200
80/80 [==============================] - 5s 66ms/step - loss: 0.4444 - accuracy:
0.8370 - val_loss: 1.3290 - val_accuracy: 0.6320
Epoch 103/200
80/80 [==============================] - 6s 80ms/step - loss: 0.4407 - accuracy:
0.8396 - val_loss: 1.3044 - val_accuracy: 0.6370
Epoch 104/200
80/80 [==============================] - 6s 75ms/step - loss: 0.4428 - accuracy:
0.8444 - val_loss: 1.4132 - val_accuracy: 0.6315
Epoch 105/200
80/80 [==============================] - 7s 88ms/step - loss: 0.4450 - accuracy:
0.8439 - val_loss: 1.3715 - val_accuracy: 0.6295
Epoch 106/200
80/80 [==============================] - 5s 68ms/step - loss: 0.4192 - accuracy:
0.8487 - val_loss: 1.4316 - val_accuracy: 0.6240
Epoch 107/200
80/80 [==============================] - 7s 89ms/step - loss: 0.4243 - accuracy:
0.8468 - val_loss: 1.3762 - val_accuracy: 0.6325
Epoch 108/200
80/80 [==============================] - 9s 105ms/step - loss: 0.4165 -
accuracy: 0.8493 - val_loss: 1.3933 - val_accuracy: 0.6445
Epoch 109/200
80/80 [==============================] - 6s 71ms/step - loss: 0.4206 - accuracy:
0.8468 - val_loss: 1.4503 - val_accuracy: 0.6280
Epoch 110/200
80/80 [==============================] - 6s 75ms/step - loss: 0.4316 - accuracy:
0.8454 - val_loss: 1.5251 - val_accuracy: 0.6150
Epoch 111/200
80/80 [==============================] - 8s 87ms/step - loss: 0.4342 - accuracy:
0.8464 - val_loss: 1.3284 - val_accuracy: 0.6385
Epoch 112/200
80/80 [==============================] - 8s 96ms/step - loss: 0.4225 - accuracy:
0.8508 - val_loss: 1.3096 - val_accuracy: 0.6505
Epoch 113/200
80/80 [==============================] - 6s 81ms/step - loss: 0.3964 - accuracy:
0.8577 - val_loss: 1.3072 - val_accuracy: 0.6500
Epoch 114/200
80/80 [==============================] - 7s 91ms/step - loss: 0.4186 - accuracy:
0.8522 - val_loss: 1.3530 - val_accuracy: 0.6650
Epoch 115/200
80/80 [==============================] - 7s 81ms/step - loss: 0.3981 - accuracy:
0.8594 - val_loss: 1.3616 - val_accuracy: 0.6500
Epoch 116/200
80/80 [==============================] - 6s 74ms/step - loss: 0.3946 - accuracy:
```

```
0.8621 - val_loss: 1.4903 - val_accuracy: 0.6300
Epoch 117/200
80/80 [==============================] - 7s 82ms/step - loss: 0.4119 - accuracy:
0.8493 - val_loss: 1.4431 - val_accuracy: 0.6370
Epoch 118/200
80/80 [==============================] - 6s 75ms/step - loss: 0.3973 - accuracy:
0.8577 - val_loss: 1.4353 - val_accuracy: 0.6365
Epoch 119/200
80/80 [==============================] - 7s 83ms/step - loss: 0.4078 - accuracy:
0.8596 - val_loss: 1.3839 - val_accuracy: 0.6395
Epoch 120/200
80/80 [==============================] - 7s 84ms/step - loss: 0.4052 - accuracy:
0.8572 - val_loss: 1.4107 - val_accuracy: 0.6205
Epoch 121/200
80/80 [==============================] - 5s 67ms/step - loss: 0.3977 - accuracy:
0.8597 - val_loss: 1.3537 - val_accuracy: 0.6525
Epoch 122/200
80/80 [==============================] - 7s 83ms/step - loss: 0.3891 - accuracy:
0.8577 - val_loss: 1.5142 - val_accuracy: 0.6330
Epoch 123/200
80/80 [==============================] - 6s 72ms/step - loss: 0.4029 - accuracy:
0.8574 - val_loss: 1.3407 - val_accuracy: 0.6655
Epoch 124/200
80/80 [==============================] - 6s 76ms/step - loss: 0.3718 - accuracy:
0.8656 - val_loss: 1.4297 - val_accuracy: 0.6420
Epoch 125/200
80/80 [==============================] - 6s 75ms/step - loss: 0.3808 - accuracy:
0.8626 - val_loss: 1.4157 - val_accuracy: 0.6350
Epoch 126/200
80/80 [==============================] - 8s 100ms/step - loss: 0.3825 -
accuracy: 0.8665 - val_loss: 1.3674 - val_accuracy: 0.6520
Epoch 127/200
80/80 [==============================] - 7s 88ms/step - loss: 0.3613 - accuracy:
0.8691 - val_loss: 1.4306 - val_accuracy: 0.6295
Epoch 128/200
80/80 [==============================] - 6s 64ms/step - loss: 0.3796 - accuracy:
0.8635 - val_loss: 1.4001 - val_accuracy: 0.6475
Epoch 129/200
80/80 [==============================] - 7s 88ms/step - loss: 0.3865 - accuracy:
0.8660 - val_loss: 1.3553 - val_accuracy: 0.6535
Epoch 130/200
80/80 [==============================] - 7s 92ms/step - loss: 0.3832 - accuracy:
0.8677 - val_loss: 1.4503 - val_accuracy: 0.6400
Epoch 131/200
80/80 [==============================] - 6s 76ms/step - loss: 0.3820 - accuracy:
0.8626 - val_loss: 1.3977 - val_accuracy: 0.6455
Epoch 132/200
80/80 [==============================] - 7s 83ms/step - loss: 0.3689 - accuracy:
```

```
0.8666 - val_loss: 1.3832 - val_accuracy: 0.6470
Epoch 133/200
80/80 [==============================] - 6s 75ms/step - loss: 0.3739 - accuracy:
0.8695 - val_loss: 1.4672 - val_accuracy: 0.6345
Epoch 134/200
80/80 [==============================] - 7s 89ms/step - loss: 0.3646 - accuracy:
0.8659 - val_loss: 1.4583 - val_accuracy: 0.6440
Epoch 135/200
80/80 [==============================] - 5s 58ms/step - loss: 0.3636 - accuracy:
0.8702 - val_loss: 1.3348 - val_accuracy: 0.6575
Epoch 136/200
80/80 [==============================] - 7s 87ms/step - loss: 0.3554 - accuracy:
0.8759 - val_loss: 1.3690 - val_accuracy: 0.6600
Epoch 137/200
80/80 [==============================] - 7s 86ms/step - loss: 0.3623 - accuracy:
0.8754 - val_loss: 1.4321 - val_accuracy: 0.6435
Epoch 138/200
80/80 [==============================] - 7s 84ms/step - loss: 0.3718 - accuracy:
0.8661 - val_loss: 1.4024 - val_accuracy: 0.6585
Epoch 139/200
80/80 [==============================] - 7s 86ms/step - loss: 0.3425 - accuracy:
0.8769 - val_loss: 1.4831 - val_accuracy: 0.6380
Epoch 140/200
80/80 [==============================] - 7s 83ms/step - loss: 0.3745 - accuracy:
0.8671 - val_loss: 1.4340 - val_accuracy: 0.6440
Epoch 141/200
80/80 [==============================] - 7s 82ms/step - loss: 0.3320 - accuracy:
0.8826 - val_loss: 1.4786 - val_accuracy: 0.6540
Epoch 142/200
80/80 [==============================] - 7s 92ms/step - loss: 0.3635 - accuracy:
0.8733 - val_loss: 1.3898 - val_accuracy: 0.6490
Epoch 143/200
80/80 [==============================] - 6s 77ms/step - loss: 0.3643 - accuracy:
0.8729 - val_loss: 1.3945 - val_accuracy: 0.6415
Epoch 144/200
80/80 [==============================] - 7s 87ms/step - loss: 0.3452 - accuracy:
0.8774 - val_loss: 1.5554 - val_accuracy: 0.6345
Epoch 145/200
80/80 [==============================] - 7s 89ms/step - loss: 0.3551 - accuracy:
0.8710 - val_loss: 1.4836 - val_accuracy: 0.6445
Epoch 146/200
80/80 [==============================] - 6s 77ms/step - loss: 0.3400 - accuracy:
0.8770 - val_loss: 1.4795 - val_accuracy: 0.6515
Epoch 147/200
80/80 [==============================] - 6s 71ms/step - loss: 0.3296 - accuracy:
0.8792 - val_loss: 1.5201 - val_accuracy: 0.6260
Epoch 148/200
80/80 [==============================] - 7s 90ms/step - loss: 0.3519 - accuracy:
```

```
0.8724 - val_loss: 1.4780 - val_accuracy: 0.6565
Epoch 149/200
80/80 [==============================] - 7s 78ms/step - loss: 0.3420 - accuracy:
0.8809 - val_loss: 1.4516 - val_accuracy: 0.6525
Epoch 150/200
80/80 [==============================] - 6s 76ms/step - loss: 0.3556 - accuracy:
0.8729 - val_loss: 1.5387 - val_accuracy: 0.6345
Epoch 151/200
80/80 [==============================] - 7s 83ms/step - loss: 0.3284 - accuracy:
0.8836 - val_loss: 1.4341 - val_accuracy: 0.6550
Epoch 152/200
80/80 [==============================] - 6s 77ms/step - loss: 0.3397 - accuracy:
0.8783 - val_loss: 1.4903 - val_accuracy: 0.6470
Epoch 153/200
80/80 [==============================] - 6s 80ms/step - loss: 0.3425 - accuracy:
0.8798 - val_loss: 1.5229 - val_accuracy: 0.6300
Epoch 154/200
80/80 [==============================] - 5s 66ms/step - loss: 0.3305 - accuracy:
0.8808 - val_loss: 1.5820 - val_accuracy: 0.6300
Epoch 155/200
80/80 [==============================] - 6s 80ms/step - loss: 0.3351 - accuracy:
0.8786 - val_loss: 1.5681 - val_accuracy: 0.6215
Epoch 156/200
80/80 [==============================] - 6s 77ms/step - loss: 0.3299 - accuracy:
0.8821 - val_loss: 1.4941 - val_accuracy: 0.6340
Epoch 157/200
80/80 [==============================] - 6s 73ms/step - loss: 0.3400 - accuracy:
0.8795 - val_loss: 1.4747 - val_accuracy: 0.6490
Epoch 158/200
80/80 [==============================] - 7s 85ms/step - loss: 0.3069 - accuracy:
0.8899 - val_loss: 1.5493 - val_accuracy: 0.6665
Epoch 159/200
80/80 [==============================] - 8s 105ms/step - loss: 0.3333 -
accuracy: 0.8795 - val_loss: 1.5370 - val_accuracy: 0.6485
Epoch 160/200
80/80 [==============================] - 6s 72ms/step - loss: 0.3182 - accuracy:
0.8832 - val_loss: 1.5347 - val_accuracy: 0.6420
Epoch 161/200
80/80 [==============================] - 6s 73ms/step - loss: 0.3439 - accuracy:
0.8819 - val_loss: 1.4082 - val_accuracy: 0.6540
Epoch 162/200
80/80 [==============================] - 6s 79ms/step - loss: 0.3191 - accuracy:
0.8860 - val_loss: 1.6004 - val_accuracy: 0.6350
Epoch 163/200
80/80 [==============================] - 6s 80ms/step - loss: 0.3336 - accuracy:
0.8832 - val_loss: 1.5578 - val_accuracy: 0.6320
Epoch 164/200
80/80 [==============================] - 7s 87ms/step - loss: 0.3231 - accuracy:
```

```
0.8824 - val_loss: 1.5614 - val_accuracy: 0.6515
Epoch 165/200
80/80 [==============================] - 7s 92ms/step - loss: 0.3358 - accuracy:
0.8829 - val_loss: 1.4627 - val_accuracy: 0.6375
Epoch 166/200
80/80 [==============================] - 9s 109ms/step - loss: 0.3085 -
accuracy: 0.8884 - val_loss: 1.6511 - val_accuracy: 0.6155
Epoch 167/200
80/80 [==============================] - 7s 86ms/step - loss: 0.3283 - accuracy:
0.8825 - val_loss: 1.5100 - val_accuracy: 0.6490
Epoch 168/200
80/80 [==============================] - 7s 81ms/step - loss: 0.3026 - accuracy:
0.8926 - val_loss: 1.5070 - val_accuracy: 0.6400
Epoch 169/200
80/80 [==============================] - 7s 85ms/step - loss: 0.3180 - accuracy:
0.8857 - val_loss: 1.5426 - val_accuracy: 0.6350
Epoch 170/200
80/80 [==============================] - 6s 70ms/step - loss: 0.2984 - accuracy:
0.8956 - val_loss: 1.5433 - val_accuracy: 0.6555
Epoch 171/200
80/80 [==============================] - 6s 71ms/step - loss: 0.2975 - accuracy:
0.8954 - val_loss: 1.4874 - val_accuracy: 0.6485
Epoch 172/200
80/80 [==============================] - 7s 88ms/step - loss: 0.2946 - accuracy:
0.8951 - val_loss: 1.5929 - val_accuracy: 0.6395
Epoch 173/200
80/80 [==============================] - 6s 78ms/step - loss: 0.3282 - accuracy:
0.8815 - val_loss: 1.5878 - val_accuracy: 0.6170
Epoch 174/200
80/80 [==============================] - 6s 72ms/step - loss: 0.2904 - accuracy:
0.8951 - val_loss: 1.6547 - val_accuracy: 0.6340
Epoch 175/200
80/80 [==============================] - 6s 81ms/step - loss: 0.3117 - accuracy:
0.8894 - val_loss: 1.6652 - val_accuracy: 0.6285
Epoch 176/200
80/80 [==============================] - 5s 66ms/step - loss: 0.3059 - accuracy:
0.8882 - val_loss: 1.5479 - val_accuracy: 0.6500
Epoch 177/200
80/80 [==============================] - 7s 84ms/step - loss: 0.2959 - accuracy:
0.8940 - val_loss: 1.5989 - val_accuracy: 0.6355
Epoch 178/200
80/80 [==============================] - 7s 81ms/step - loss: 0.2855 - accuracy:
0.8986 - val_loss: 1.5935 - val_accuracy: 0.6515
Epoch 179/200
80/80 [==============================] - 6s 76ms/step - loss: 0.3006 - accuracy:
0.8940 - val_loss: 1.5863 - val_accuracy: 0.6330
Epoch 180/200
80/80 [==============================] - 7s 90ms/step - loss: 0.3004 - accuracy:
```

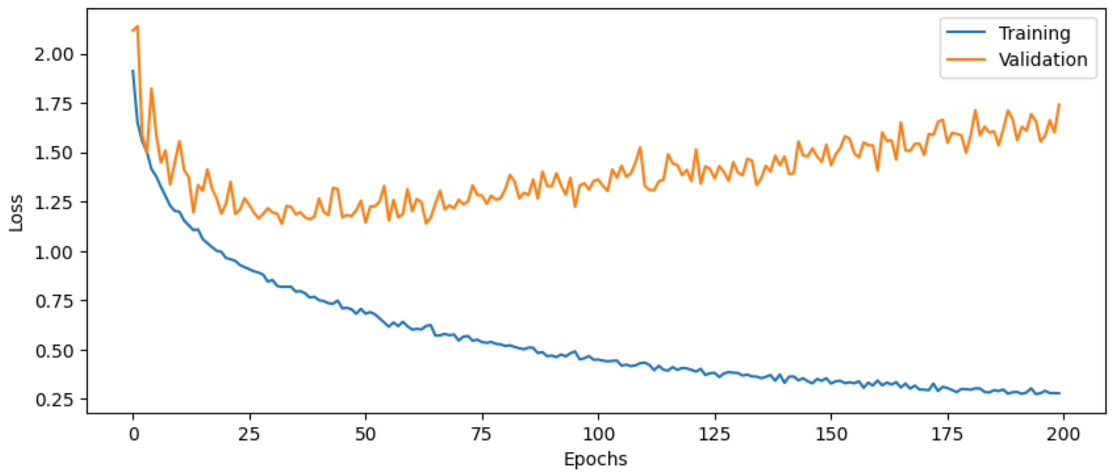
```
0.8910 - val_loss: 1.4976 - val_accuracy: 0.6560
Epoch 181/200
80/80 [==============================] - 5s 66ms/step - loss: 0.2972 - accuracy:
0.8928 - val_loss: 1.5846 - val_accuracy: 0.6460
Epoch 182/200
80/80 [==============================] - 7s 74ms/step - loss: 0.3040 - accuracy:
0.8909 - val_loss: 1.7139 - val_accuracy: 0.6310
Epoch 183/200
80/80 [==============================] - 7s 82ms/step - loss: 0.3033 - accuracy:
0.8936 - val_loss: 1.5860 - val_accuracy: 0.6505
Epoch 184/200
80/80 [==============================] - 6s 77ms/step - loss: 0.2857 - accuracy:
0.8988 - val_loss: 1.6300 - val_accuracy: 0.6335
Epoch 185/200
80/80 [==============================] - 6s 78ms/step - loss: 0.2834 - accuracy:
0.8989 - val_loss: 1.6001 - val_accuracy: 0.6405
Epoch 186/200
80/80 [==============================] - 7s 88ms/step - loss: 0.2950 - accuracy:
0.8956 - val_loss: 1.6072 - val_accuracy: 0.6400
Epoch 187/200
80/80 [==============================] - 6s 65ms/step - loss: 0.2902 - accuracy:
0.8959 - val_loss: 1.5353 - val_accuracy: 0.6445
Epoch 188/200
80/80 [==============================] - 6s 77ms/step - loss: 0.2981 - accuracy:
0.8932 - val_loss: 1.6165 - val_accuracy: 0.6405
Epoch 189/200
80/80 [==============================] - 7s 88ms/step - loss: 0.2774 - accuracy:
0.9006 - val_loss: 1.7117 - val_accuracy: 0.6450
Epoch 190/200
80/80 [==============================] - 8s 89ms/step - loss: 0.2844 - accuracy:
0.8947 - val_loss: 1.6685 - val_accuracy: 0.6480
Epoch 191/200
80/80 [==============================] - 7s 89ms/step - loss: 0.2857 - accuracy:
0.8917 - val_loss: 1.5613 - val_accuracy: 0.6485
Epoch 192/200
80/80 [==============================] - 8s 98ms/step - loss: 0.2766 - accuracy:
0.9019 - val_loss: 1.6297 - val_accuracy: 0.6395
Epoch 193/200
80/80 [==============================] - 6s 71ms/step - loss: 0.2831 - accuracy:
0.8974 - val_loss: 1.6094 - val_accuracy: 0.6390
Epoch 194/200
80/80 [==============================] - 7s 85ms/step - loss: 0.3041 - accuracy:
0.8942 - val_loss: 1.6930 - val_accuracy: 0.6230
Epoch 195/200
80/80 [==============================] - 7s 82ms/step - loss: 0.2750 - accuracy:
0.9010 - val_loss: 1.6589 - val_accuracy: 0.6390
Epoch 196/200
80/80 [==============================] - 6s 75ms/step - loss: 0.2794 - accuracy:
```

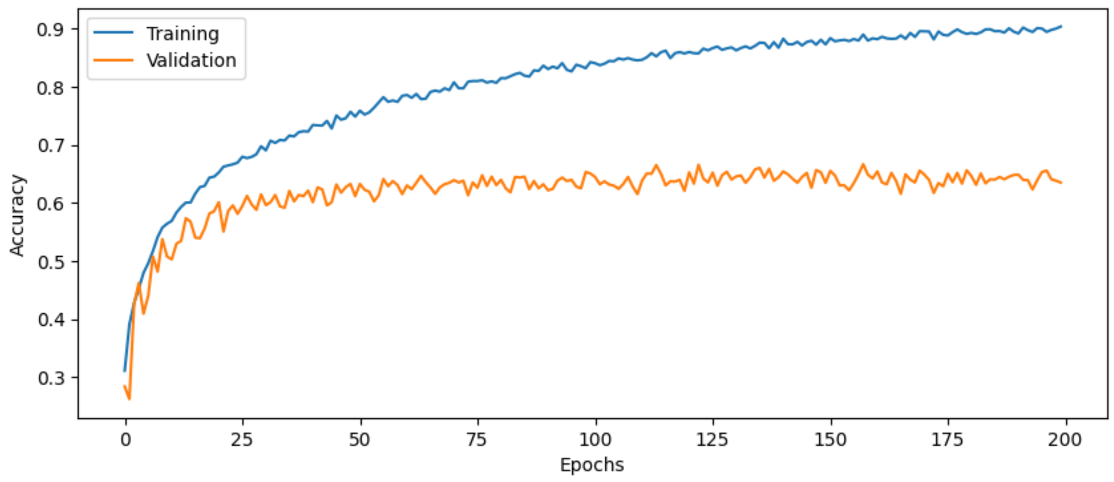
```
0.9001 - val_loss: 1.5542 - val_accuracy: 0.6530
Epoch 197/200
80/80 [==============================] - 7s 88ms/step - loss: 0.2919 - accuracy:
0.8944 - val_loss: 1.5845 - val_accuracy: 0.6555
Epoch 198/200
80/80 [==============================] - 7s 75ms/step - loss: 0.2804 - accuracy:
0.8979 - val_loss: 1.6628 - val_accuracy: 0.6405
Epoch 199/200
80/80 [==============================] - 6s 72ms/step - loss: 0.2802 - accuracy:
0.9001 - val_loss: 1.6007 - val_accuracy: 0.6380
Epoch 200/200
80/80 [==============================] - 6s 81ms/step - loss: 0.2790 - accuracy:
0.9036 - val_loss: 1.7412 - val_accuracy: 0.6350
```

[ ]:
```
# 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, Ytestc, 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, Ytestc, batch_size = batch_size,␣
↪verbose=0)
print('Test loss: %.4f' % score[0])
print('Test accuracy: %.4f' % score[1])
```

```
Test loss: 1.6627
Test accuracy: 0.6560
Test loss: 4.3088
Test accuracy: 0.3215
```

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





3.13 Part 20: Plot misclassified images

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

[ ]:
```
# 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 [==============================] - 0s 2ms/step
```

[ ]:

```python
# 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()
```

plane, classified as car
plane, classified as horse
plane, classified as bird
plane, classified as cat
plane, classified as truck
plane, classified as car
plane, classified as cat
plane, classified as truck
plane, classified as deer
plane, classified as horse
plane, classified as frog
plane, classified as bird
plane, classified as frog
plane, classified as bird
plane, classified as horse
plane, classified as dog
plane, classified as truck
plane, classified as frog

3.14 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?

3.15 Answer to question 25

No, The reason why we can not use different size images as input is mainly because the dense layers are expected to accept fixed-size images

3.16 Answer to question 26

Yes, we can resize the input image to the expected size.

By using tensorflow. keras.preprocessing image, we can load the image and resize it to the size that we need.

Meanwhile, we can use full connected network since it does not have any dense layers.

3.17 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

3.18 Answer to question 27

ResNet50 has 50-layer convolutional neural network (48 convolutional layers, one MaxPool layer, and one average pool layer).

3.19 Answer to question 28

ResNet-50 contains around 25.6 million parameters including over 23 million trainable parameters.

3.20 Answer to question 29

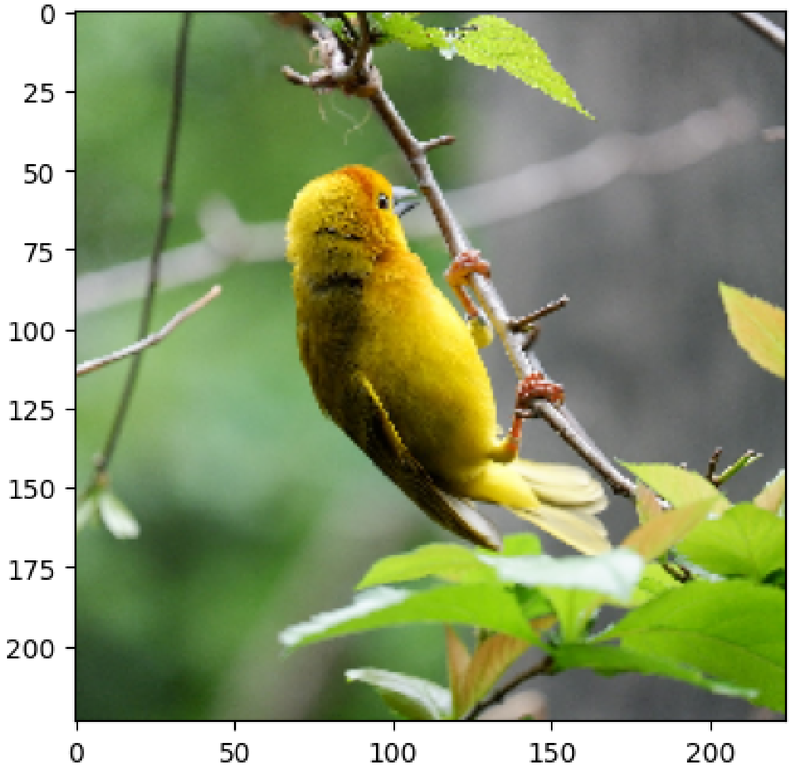
Resnet-50 requires input images to be of size 224×224

3.21 Answer to question 30

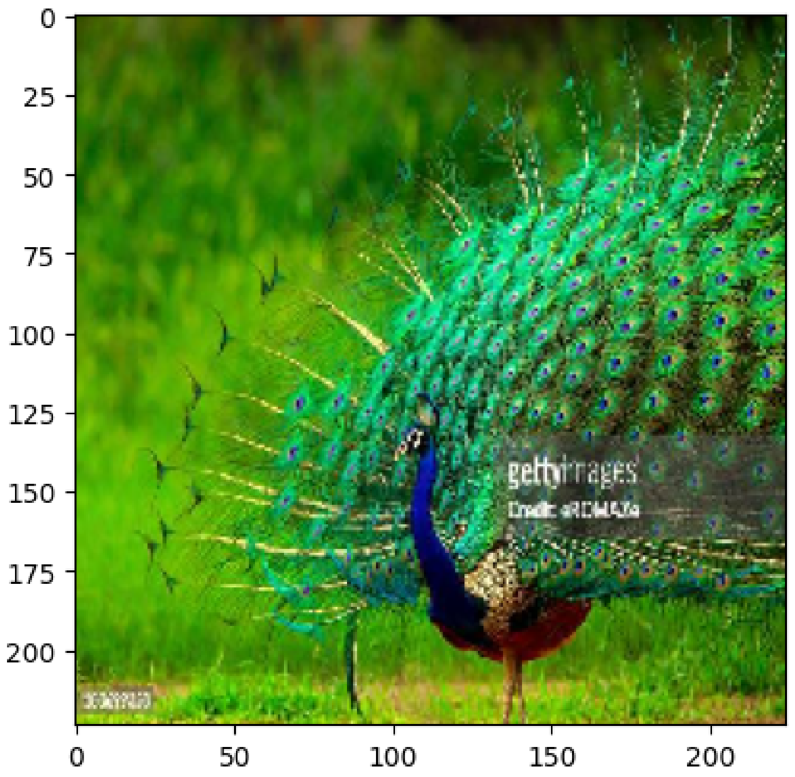
Calculating and storing the Hessian matrix can be computationally expensive, especially for large networks with millions of parameters. Use ResNet-50 as example, it contains around 25.6 million parameters, and it will take long time to calculate second derivative.

```
[ ]: # 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.

     import numpy as np
     import matplotlib.pyplot as plt
     from tensorflow.keras.applications.resnet50 import ResNet50, preprocess_input,␣
      ↪decode_predictions
     from tensorflow.keras.preprocessing import image

     images_folder = "images"
     images_to_load = ['1.jpg','2.jpg','3.jpg','4.jpg','5.jpg',]

     for i in range(len(images_to_load)):
         im_path = images_folder + "/" + images_to_load[i]
         im = image.load_img(im_path, target_size=(224, 224))
         inp = image.img_to_array(im)
         inp = np.expand_dims(inp, axis=0)
         inp = preprocess_input(inp)
         model_res = ResNet50()
         predLabs = decode_predictions(model_res.predict(inp))
         print(predLabs[0][0][1:3])
         plt.imshow(im)
         plt.show()
```

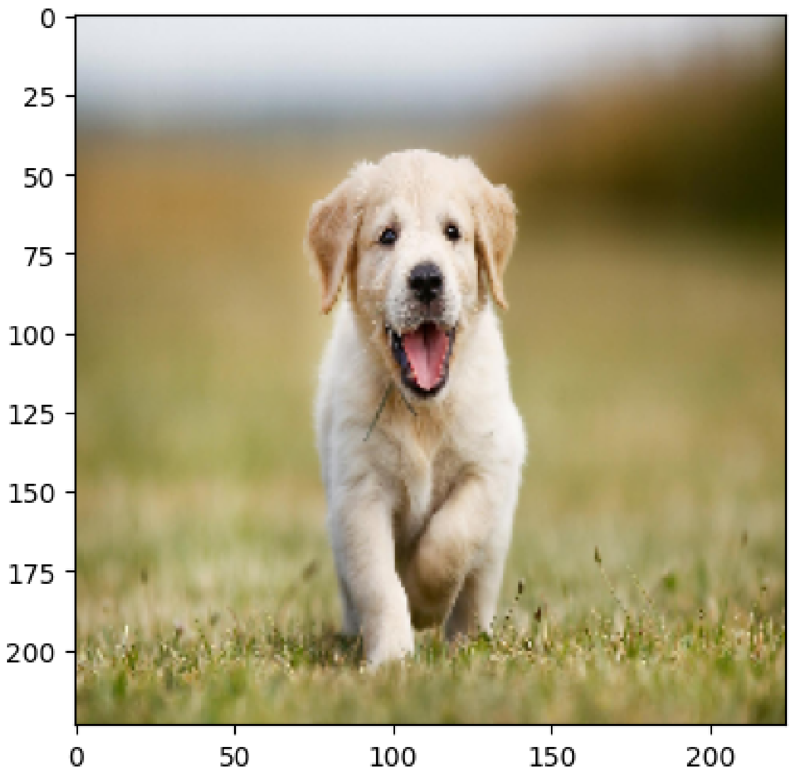
```
Downloading data from https://storage.googleapis.com/tensorflow/keras-
applications/resnet/resnet50_weights_tf_dim_ordering_tf_kernels.h5
102967424/102967424 [==============================] - 8s 0us/step
1/1 [==============================] - 3s 3s/step
Downloading data from https://storage.googleapis.com/download.tensorflow.org/dat
a/imagenet_class_index.json
35363/35363 [==============================] - 0s 3us/step
('goldfinch', 0.98532975)
```



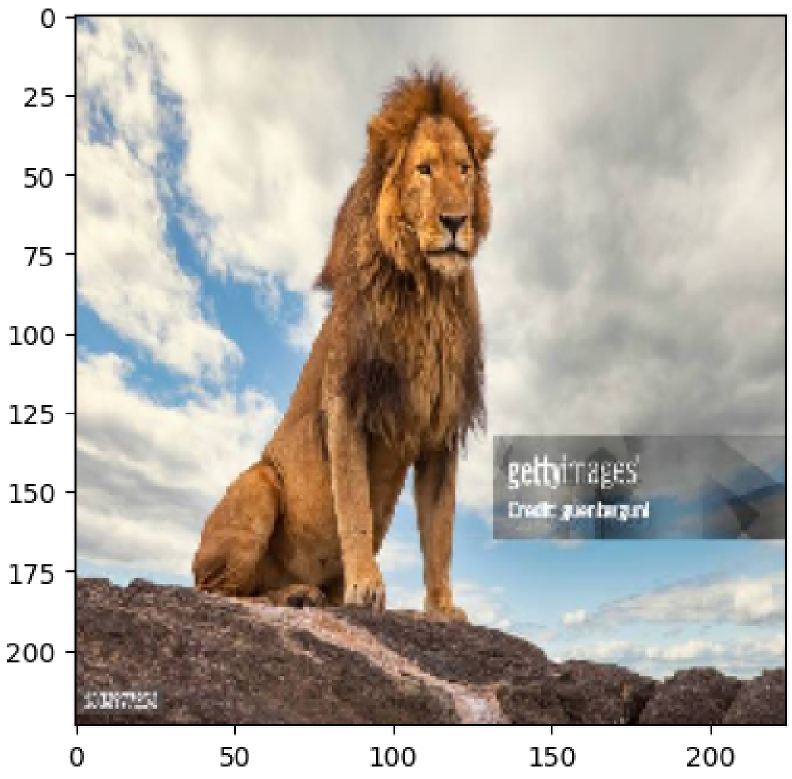
```
1/1 [==============================] - 1s 1s/step
('peacock', 0.9985185)
```



```
1/1 [==============================] - 1s 929ms/step
('golden_retriever', 0.461679)
```

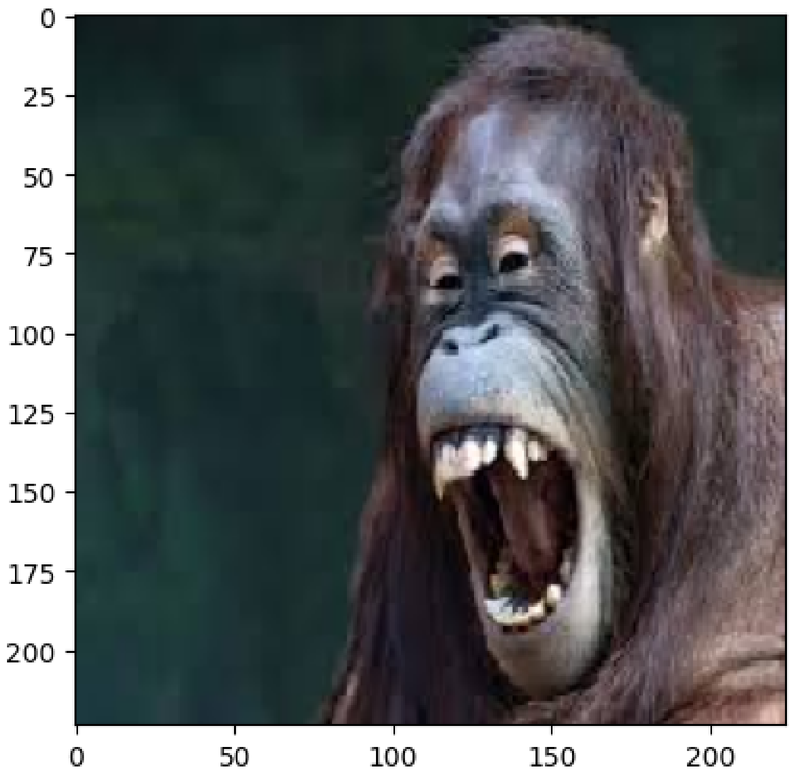


```
WARNING:tensorflow:5 out of the last 67 calls to <function
Model.make_predict_function.<locals>.predict_function at 0x7f6b104b2560>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings could be due to (1) creating @tf.function repeatedly in a loop, (2)
passing tensors with different shapes, (3) passing Python objects instead of
tensors. For (1), please define your @tf.function outside of the loop. For (2),
@tf.function has reduce_retracing=True option that can avoid unnecessary
retracing. For (3), please refer to
https://www.tensorflow.org/guide/function#controlling_retracing and
https://www.tensorflow.org/api_docs/python/tf/function for  more details.

1/1 [==============================] - 1s 1s/step
('lion', 0.8851059)
```



```
WARNING:tensorflow:6 out of the last 68 calls to <function
Model.make_predict_function.<locals>.predict_function at 0x7f6aa3749cf0>
triggered tf.function retracing. Tracing is expensive and the excessive number
of tracings could be due to (1) creating @tf.function repeatedly in a loop, (2)
passing tensors with different shapes, (3) passing Python objects instead of
tensors. For (1), please define your @tf.function outside of the loop. For (2),
@tf.function has reduce_retracing=True option that can avoid unnecessary
retracing. For (3), please refer to
https://www.tensorflow.org/guide/function#controlling_retracing and
https://www.tensorflow.org/api_docs/python/tf/function for  more details.
```

```
1/1 [==============================] - 1s 1s/step
('orangutan', 0.98837215)
```


0
25
50
75
100
125
150
175
200
0
50
100
150
200