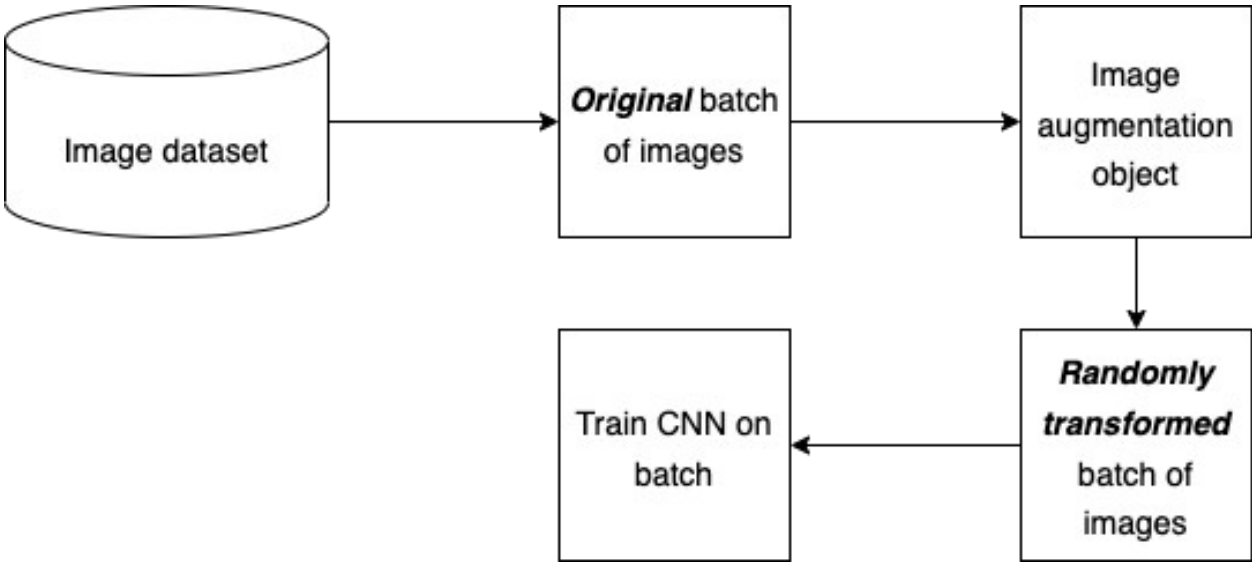
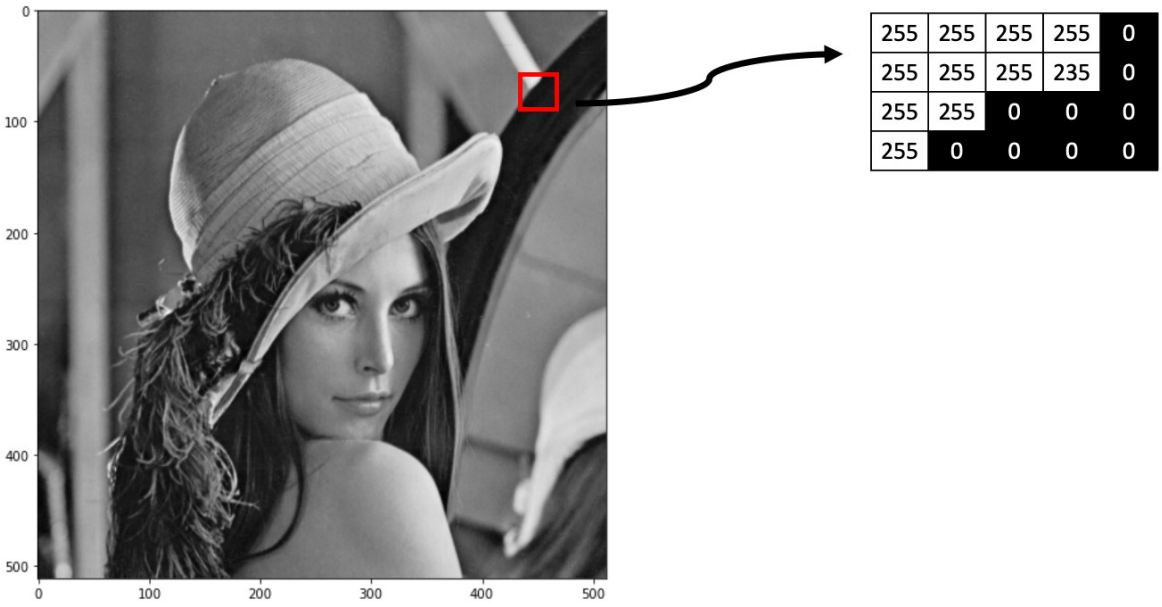
Loading Images in Keras

Convolutional neural networks (CNNs) have had great success in certain kinds of problems, such as image recognition. Data loading and preparation are important steps when it comes to working with such models. Increasingly, data augmentation is also required for more complex object recognition tasks. In this lab, we will discover various ways of loading images, as well as converting, augmenting and saving image datasets using the Keras API.

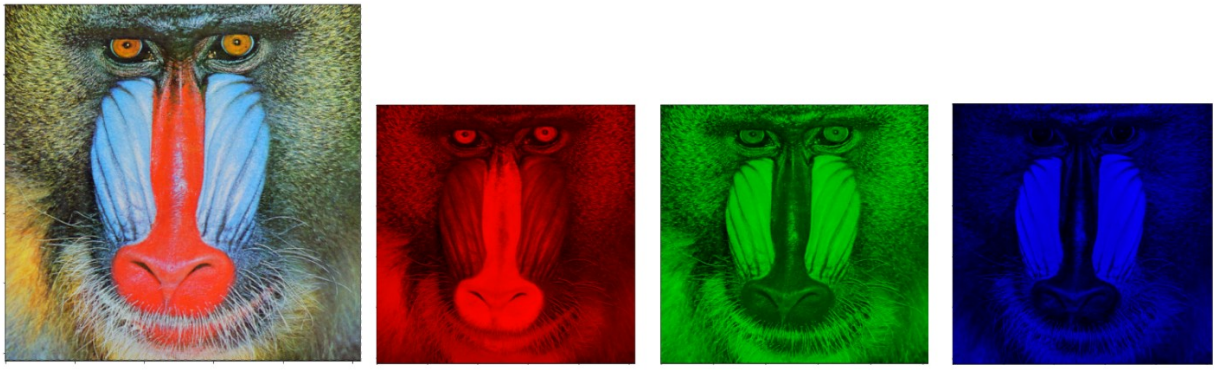


What's a Digital Image?

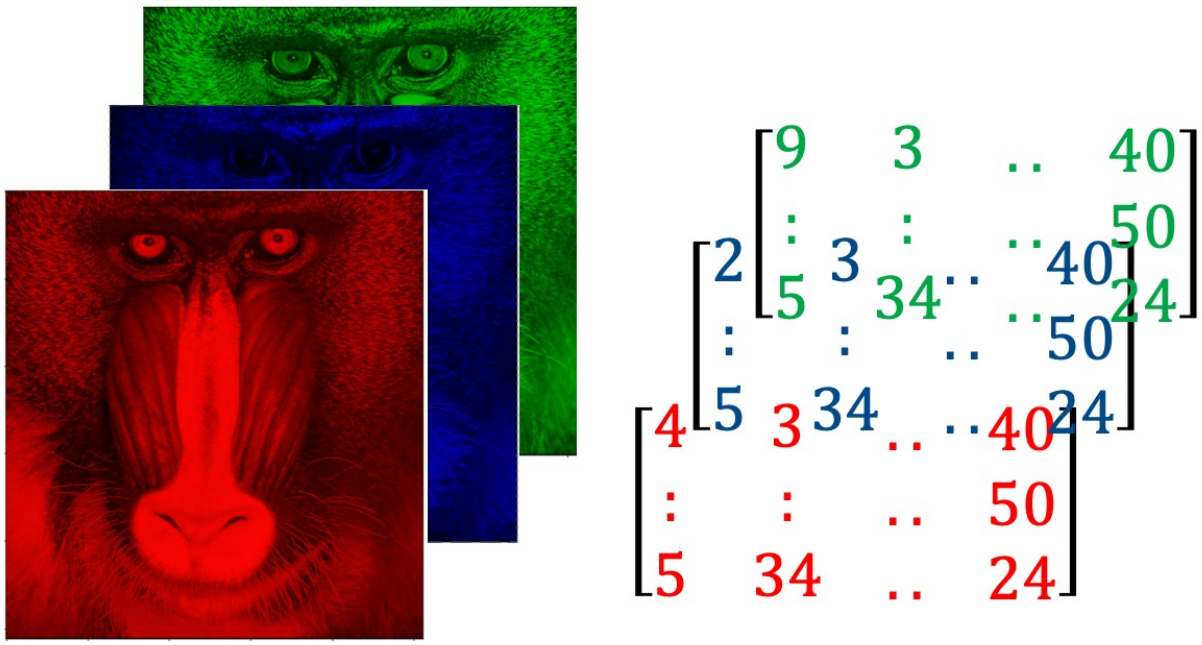
A gray-scale digital image can be interpreted as a rectangular array of numbers. If we zoom into the region, we see the image is comprised of a rectangular grid of discrete blocks called pixels. We can represent these pixels with numbers called intensity values ranging from 0 to 255, as shown here:



Color images are a combination of red, blue, and green intensity values as shown here



each channel has values ranging from 0 to 255.



Finally, to apply a neural network to Karis we sometimes add a batch dimension, this is just an extra dimension. Just think of the batch dimension as an address that contains an image array.

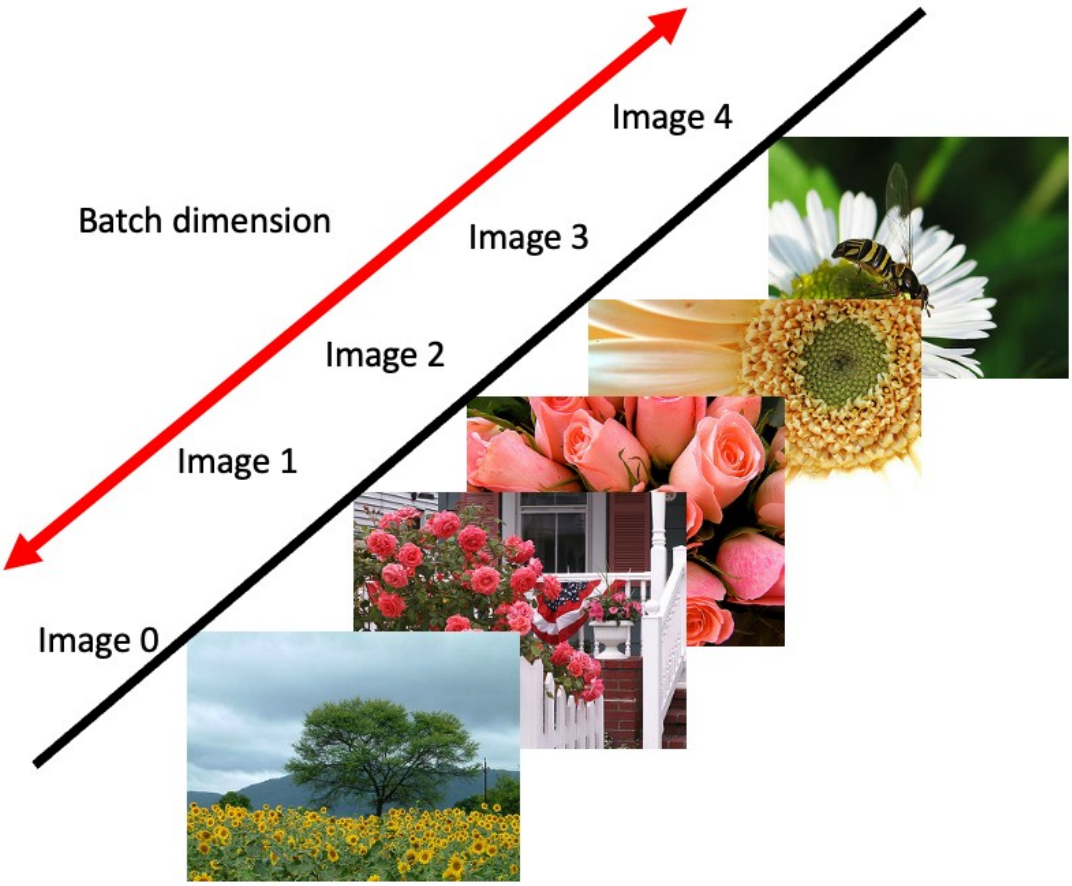


Table of Contents

After completing this lab you will be able to **apply** Keras to:

- Load and display images in multiple ways
- Convert between array and PIL formats
- Convert images to grayscale, augment images through transformations and save images to files

---

we will be using the following libraries:

- `numpy` for mathematical operations.
- `Pillow` for image processing functions.
- `OpenCV` for other image processing functions.
- `tensorflow` for machine learning and neural network related functions.
- `matplotlib` for additional plotting tools.
- `keras` for building artificial neural networks.

Installing Required Libraries

if you run these notebook commands in a different Jupyter environment (e.g. Watson Studio or Ananconda), you will need to install these libraries by removing the `#` sign before `!mamba` in the code cell below.

```
#!pip install numpy==1.22.3 matplotlib==3.5.1 tensorflow==2.9.0
opencv-python==4.5.5.62 keras

# Note: If your environment doesn't support "!mamba install", use "!
pip install --user"

# RESTART YOUR KERNEL AFTERWARD AS WELL

!pip3 install --upgrade tensorflow

# RESTART YOUR KERNEL AFTERWARD AS WELL
```

Importing Required Libraries

We recommend you import all required libraries in one place (here):

```
# You can also use this section to suppress warnings generated by your
code:
def warn(*args, **kwargs):
    pass


import warnings
warnings.warn = warn
warnings.filterwarnings('ignore')

import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2' # tensorflow INFO and WARNING
messages are not printed

import random

import pathlib
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

import PIL
import PIL.Image
from PIL import Image, ImageOps
import tensorflow as tf
```

```
import keras
from keras.preprocessing import image
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.preprocessing.image import load_img
from keras.models import Model
from keras.layers import Input, Dense

import cv2

sns.set_context('notebook')
sns.set_style('white')
```

Defining Helper Functions

```
# This function will allow us to visualize four sample images from the
loaded toy image dataset.
def visualize(X_train):
    plt.rcParams['figure.figsize'] = (6,6)

    for i in range(4):
        plt.subplot(2,2,i+1)
        num = random.randint(0, len(X_train))
        plt.imshow(X_train[num], cmap='gray', interpolation='none',
vmin=0, vmax=255)
        plt.title("class {}".format(y_train[num]))

    plt.tight_layout()
```

Background

Keras is an open-source Python library used for developing and evaluating deep learning models. It provides utilities for loading, preparing, converting, augmenting, and saving image data. In this lab, we will explore various ways of loading image datasets in Keras.

```
# Print tensorflow version
print(tf.__version__)

# Try !pip install --upgrade tensorflow if the version printed
# is less than 2.9.0
```

```
2.18.0
```

Ways to load images

We will look into four main ways of using image datasets in Keras:

- Loading in a ready-to-use toy dataset from Keras

- Loading individual images as PIL objects
- Reading a directory of images on disk using `tf.keras.utils.image_dataset_from_directory`.
- Loading image from URL

A. Ready-to-use toy datasets

The `tf.keras.datasets` in Keras provide a few in-built image datasets that have been cleaned and are typically helpful in debugging models, or creating simple examples.

These include MNIST hand-written digits, Fashion MNIST, CIFAR10, and CIFAR100.

MNIST hand-written digits is a collection of 60,000 28x28 grayscale images belonging to 10 different classes, along with a test set of 10,000 images. Let us load in the MNIST hand-written digits dataset using the `load_data` function.

```
(X_train, y_train), (X_test, y_test) =
keras.datasets.mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-
keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```

X_train is a uint8 numpy array of grayscale images with shapes (60000, 28, 28) from the training dataset. Its pixel values range from 0 to 255.

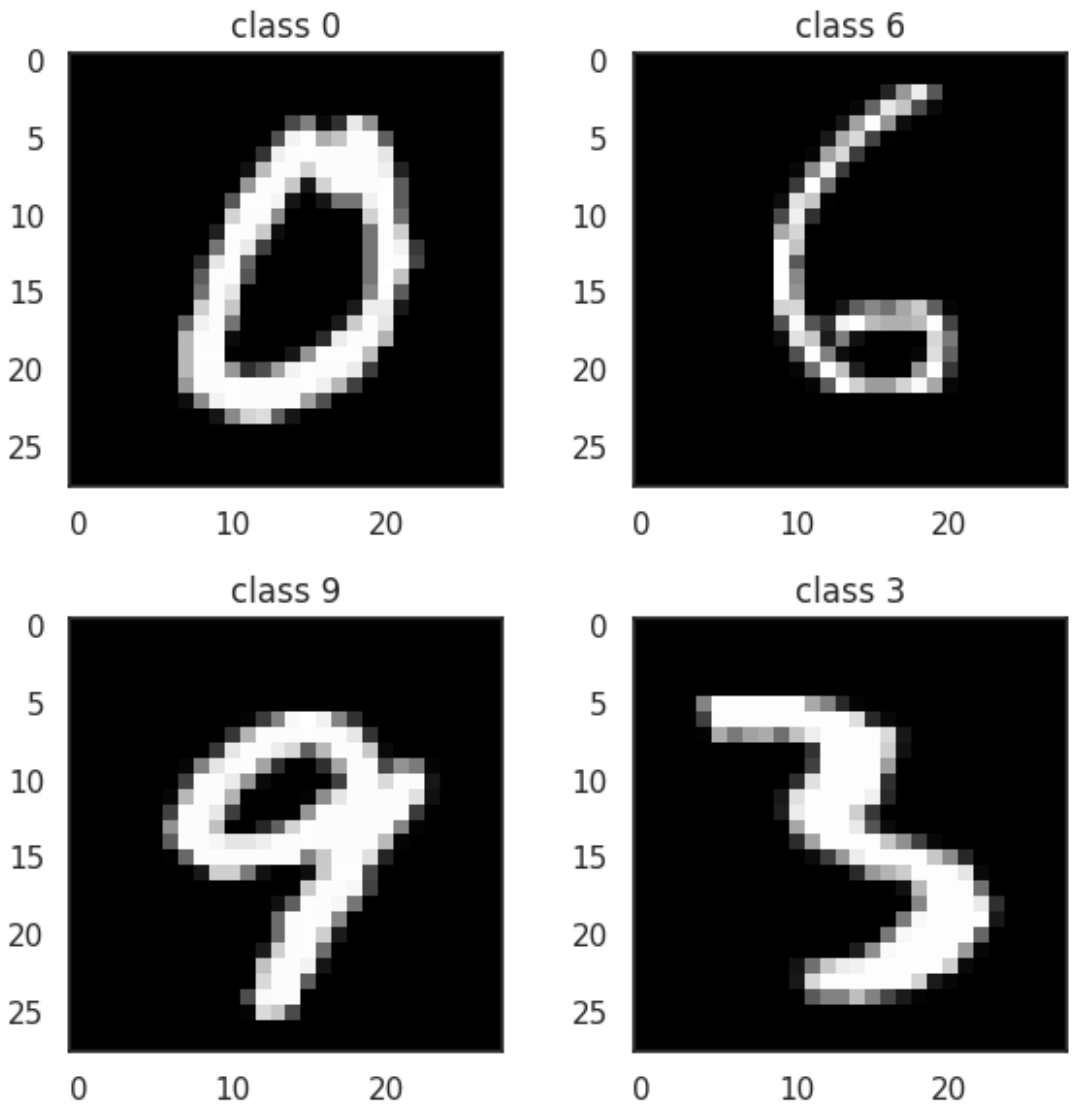
y_train is a uint8 numpy array of digit labels (integers in range 0-9) with shape (60000) for the training data.

```
print(X_train.shape)
print(X_train.dtype)
print(y_train.shape)
print(y_train.dtype)
```

```
(60000, 28, 28)
uint8
(60000,)
uint8
```

Let's visualize four random images from the training set. Only one color channel is needed to display the image as a grayscale. So for this step, we take a single color channel and display the image using the `plt.imshow()` method with `cmap` set to `gray`, `vmin` set to 0, and `vmax` set to 255.

```
visualize(X_train)
```



Using image data with Autoencoders

an autoencoder (AE) is a neural network where the input is the same as the output.

To build an AE, you need three things:

- an encoding function
- a decoding function
- distance function between the amount of information loss between the compressed representation of your data and the decompressed representation, that is, a loss function

Some practical applications of AEs are data denoising and dimensionality reduction for data visualization.

We can define a size for our encoded representations, and use `Dense()` to create encoded representation of the input. Similarly, we can use `Dense()` to create lossy reconstructions of the

input. As seen below, `encoded` has a shape of (None, 64) and `reconstruction` has the original input size of (None, 784).

```
ENCODING_DIM = 64

# Encoded representations:
inputs = Input(shape=(784,))
encoded = Dense(ENCODING_DIM, activation="sigmoid")(inputs)

# Reconstructions:
encoded_inputs = Input(shape=(ENCODING_DIM,), name='encoding')
reconstruction = Dense(784, activation="sigmoid")(encoded_inputs)

print("Encoded Input: ", encoded.shape)
print("Reconstructed Input: ", reconstruction.shape)
```

```
Encoded Input:  (None, 64)
Reconstructed Input:  (None, 784)
```

Using this technique, we can instantiate three types of models:

- End-to-end AEs mapping inputs to reconstructions
- An encoder mapping inputs to the latent space
- A decoder that takes in points from latent space and output corresponding reconstructed samples

Exercise 1 - Load Fashion MNIST

Fashion MNIST is another collection of 60,000 28x28 grayscale images belonging to 10 different classes, along with a test set of 10,000 images. Write code to load the Fashion MNIST dataset using `fashion_mnist.load_data()` from `keras.datasets`.

```
(X_train, y_train), (X_test, y_test) =
keras.datasets.fashion_mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-
keras-datasets/train-labels-idx1-ubyte.gz
29515/29515 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-
keras-datasets/train-images-idx3-ubyte.gz
26421880/26421880 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-
keras-datasets/t10k-labels-idx1-ubyte.gz
5148/5148 ━━━━━━━━━━━━━━━━━━━━ 0s 1us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-
keras-datasets/t10k-images-idx3-ubyte.gz
4422102/4422102 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```

X_train is a uint8 numpy array of grayscale images with shapes (60000, 28, 28) from the training dataset. Its pixel values range from 0 to 255.

y_train is a uint8 numpy array of digit labels (integers in range 0-9) with shape (60000,) for the training data.

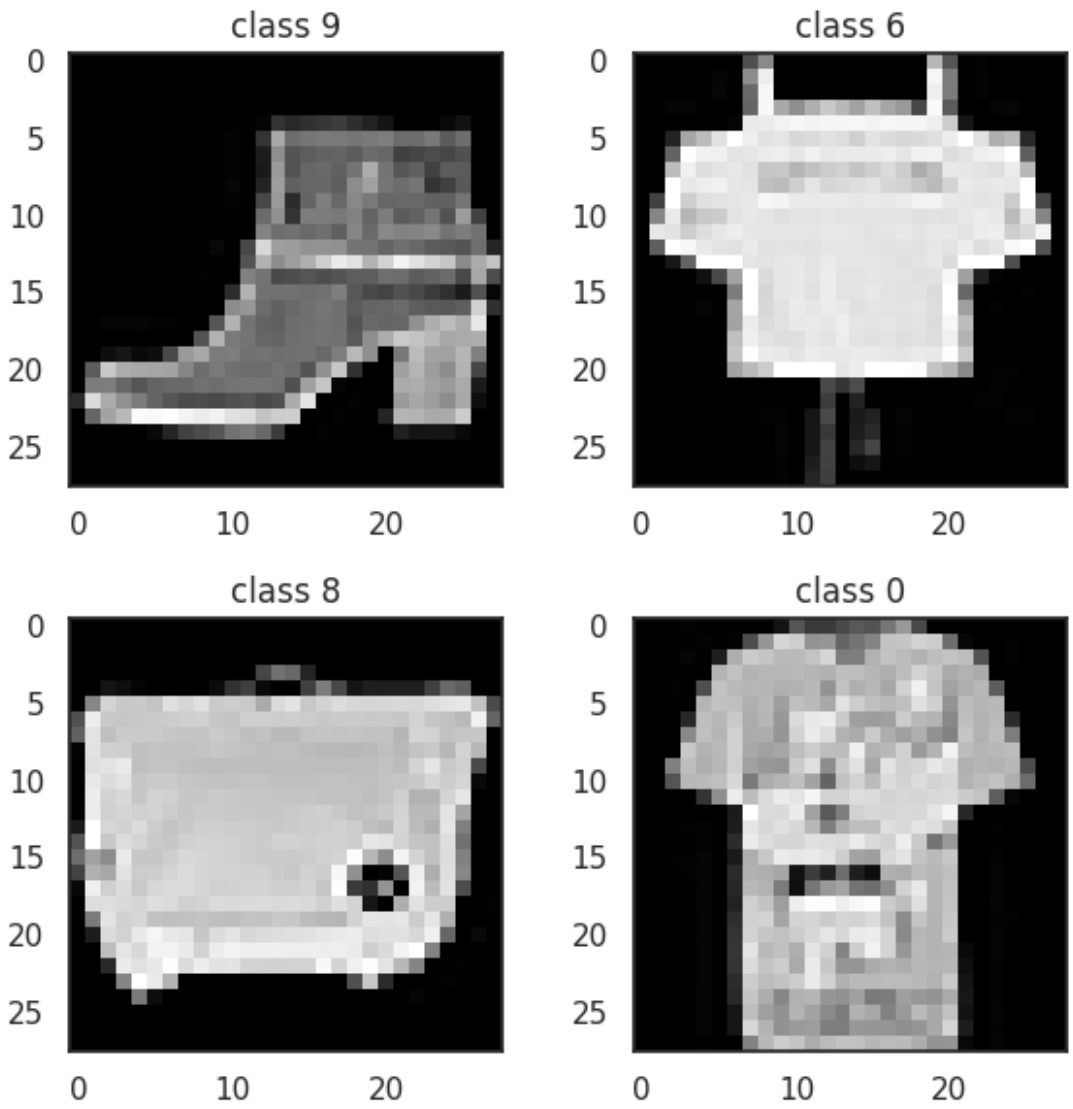
Now print the shape and dtype of the X_train and y_train arrays.

```
print(X_train.shape)
print(X_train.dtype)
print(y_train.shape)
print(y_train.dtype)
```

```
(60000, 28, 28)
uint8
(60000,)
uint8
```

Display a few randomly selected images as grayscale using the `visualize()` helper function.

```
visualize(X_train)
```



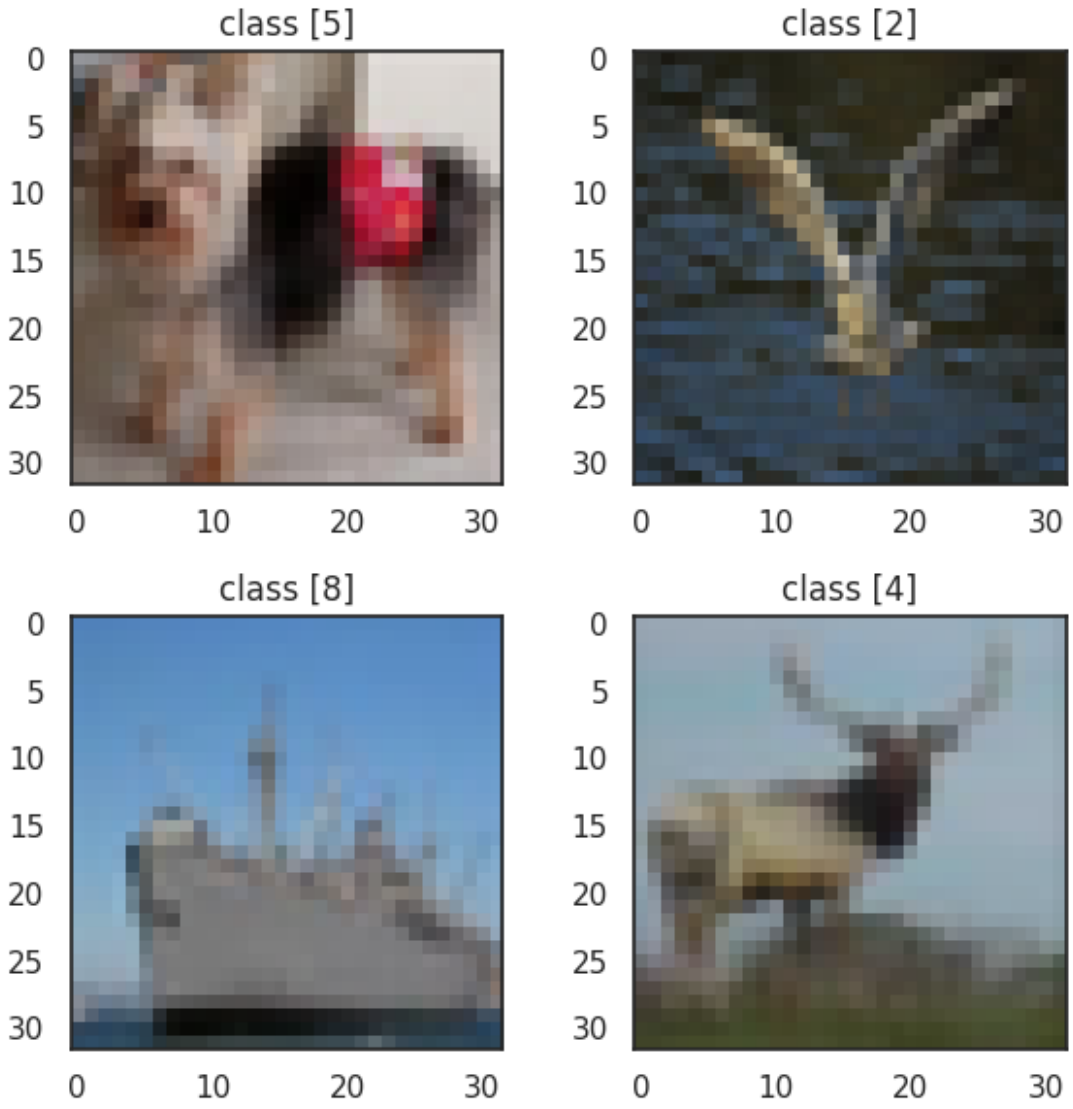
CIFAR10 is a collection of 50,000 32x32 color training images labeled over 10 different categories, along with a test set of 10,000 images.

```
(X_train, y_train), (X_test, y_test) =
keras.datasets.cifar10.load_data()

Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-
python.tar.gz
170498071/170498071 ━━━━━━━━━━━━━━━━━━━━ 3s 0us/step
```

X_train is a uint8 numpy array of grayscale image data with shapes (50000, 32, 32, 3), containing the training data. Pixel values range from 0 to 255.

y_train is a uint8 NumPy array of labels (integers in range 0-9) with shape (50000, 1) for the training data.

```
print(X_train.shape)
print(X_train.dtype)

(50000, 32, 32, 3)
uint8

visualize(X_train)
```



CIFAR100 is a collection of 50,000 32x32 color training images labeled over 100 fine-grained classes and 20 coarse-grained classes, along with a test set of 10,000 images. The 100 classes in the CIFAR-100 are grouped into 20 superclasses. Each image comes with a "fine" label (label) and a "coarse" label (superclass).

Using the fine `label_mode`:

```
(X_train, y_train), (X_test, y_test) =
keras.datasets.cifar100.load_data(label_mode = 'fine')
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-100-
python.tar.gz
169001437/169001437 ━━━━━━━━━━━━━━━━━━━━ 5s 0us/step
```

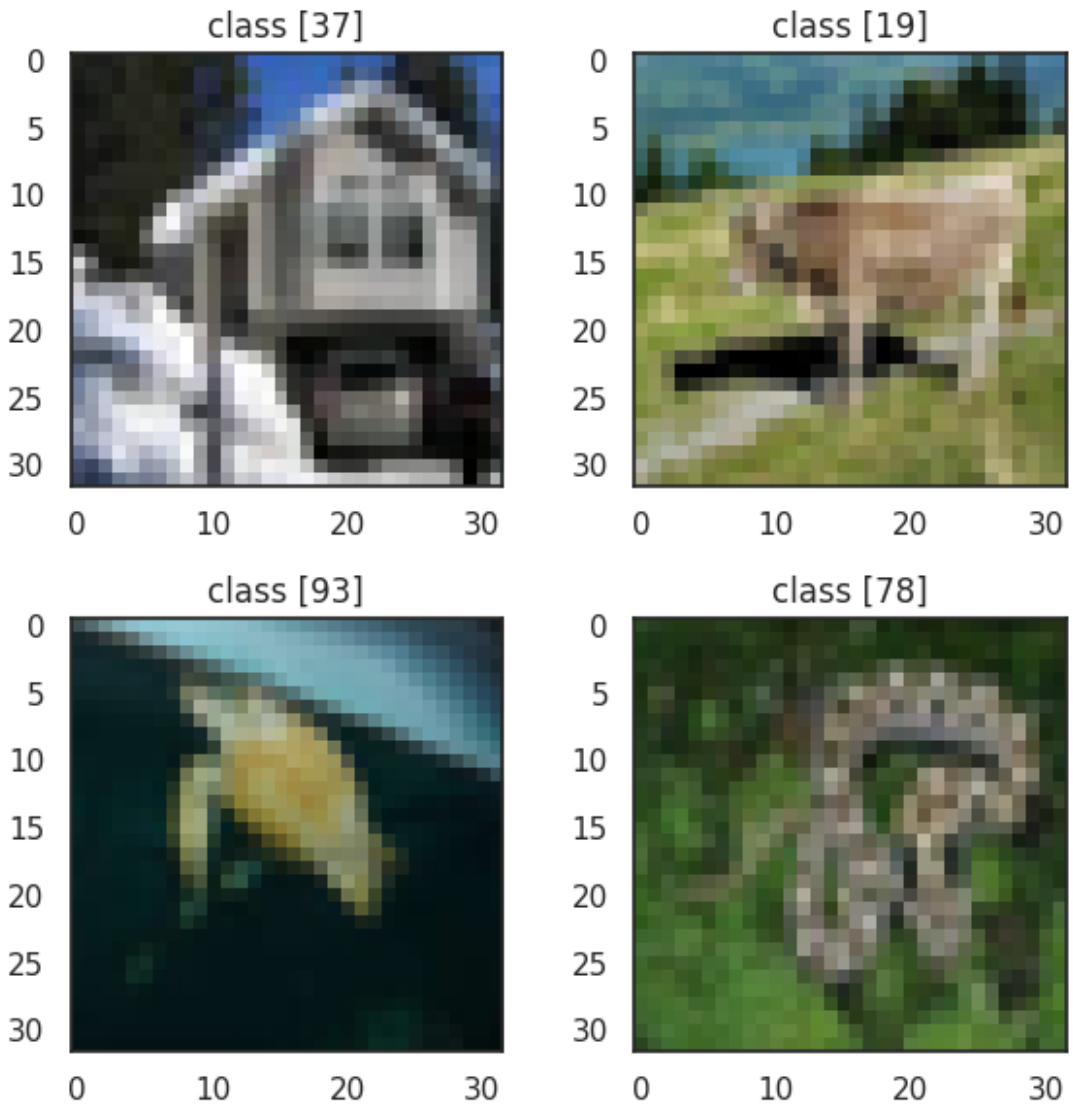
X_train is a uint8 numpy array of grayscale image data with shapes (50000, 32, 32, 3), containing the training data. Pixel values range from 0 to 255.

y_train is a uint8 numpy array of labels (integers in range 0-99) with shape (50000, 1) for the training data.

```
print(X_train.shape)
print(X_train.dtype)
```

```
(50000, 32, 32, 3)
uint8
```

```
visualize(X_train)
```



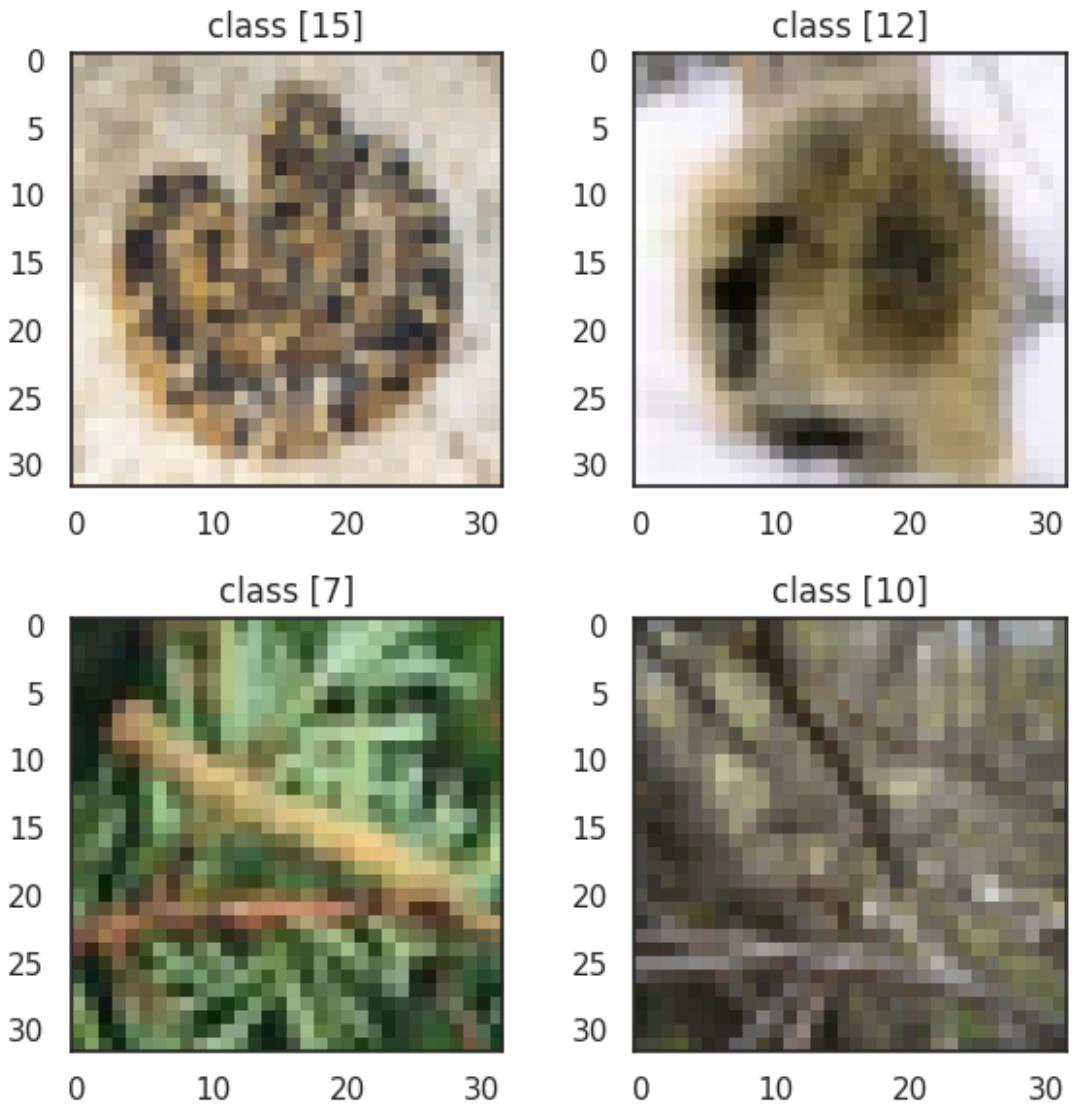
Exercise 2 - Load CIFAR100 with coarse label_mode

Load the CIFAR100 dataset by setting the pramter `label_mode = 'coarse'` in the `load_data` function .

```
(X_train, y_train), (X_test, y_test) =
keras.datasets.cifar100.load_data(label_mode = 'coarse')
```

Visualize a few images from the training set using the `visualize()` helper function.

```
visualize(X_train)
```



B. Load individual images as PIL objects

We will save the following image into our working environment.

```
import requests

URL = 'https://cf-courses-data.s3.us.cloud-object-
storage.appdomain.cloud/IBMDeveloperSkillsNetwork-ML311-Coursera/
labs/Module3/L1/Dog_Breeds.jpg'
filename = URL.split('/')[-1]
r = requests.get(URL, allow_redirects=True)
open(filename, 'wb').write(r.content)
```

```
70761
```

Next, we will specify a `target_size` of the image using a tuple of ints.

```
img_height, img_width = 100, 100
```

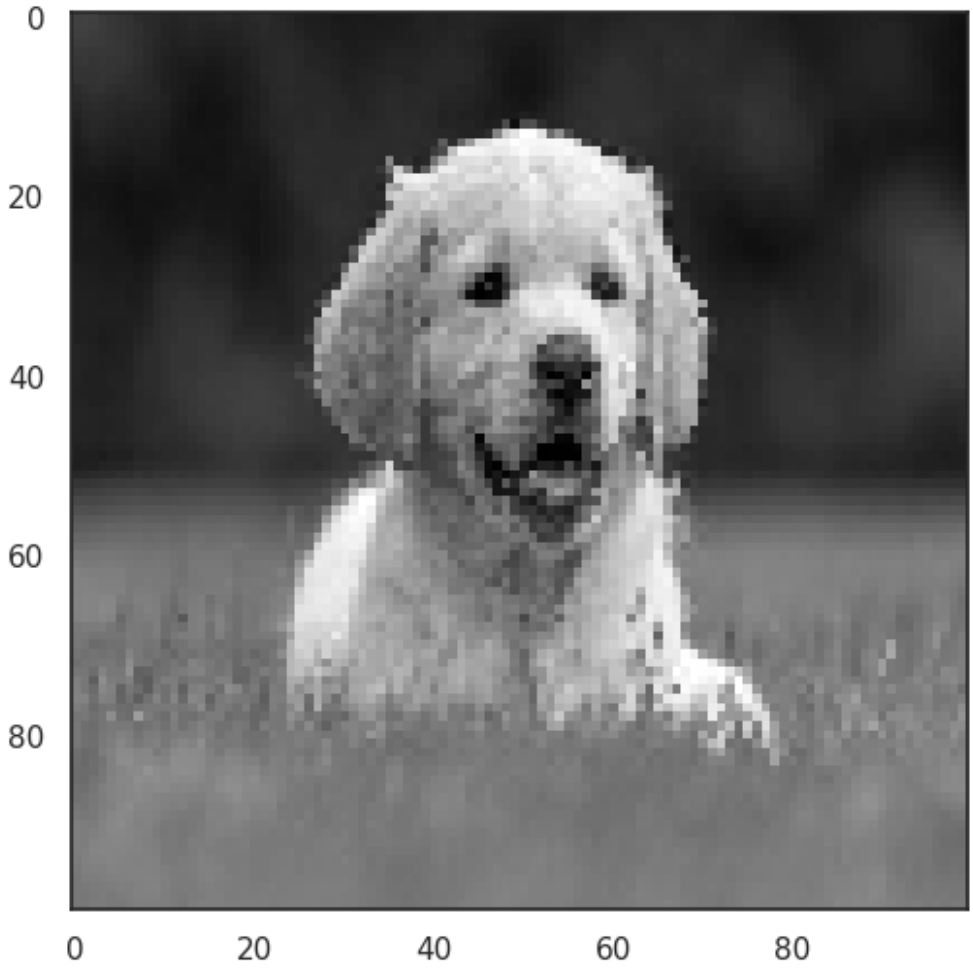
Using the path of the image we saved, we will use the `load_img` function to return a PIL Image instance.

```
gray_img = load_img(
    'Dog_Breeds.jpg',
    target_size=(img_height, img_width),
    interpolation='nearest', color_mode='grayscale')

color_img = load_img(
    'Dog_Breeds.jpg',
    target_size=(img_height, img_width),
    interpolation='nearest')


print(type(gray_img))
print(type(color_img))
```

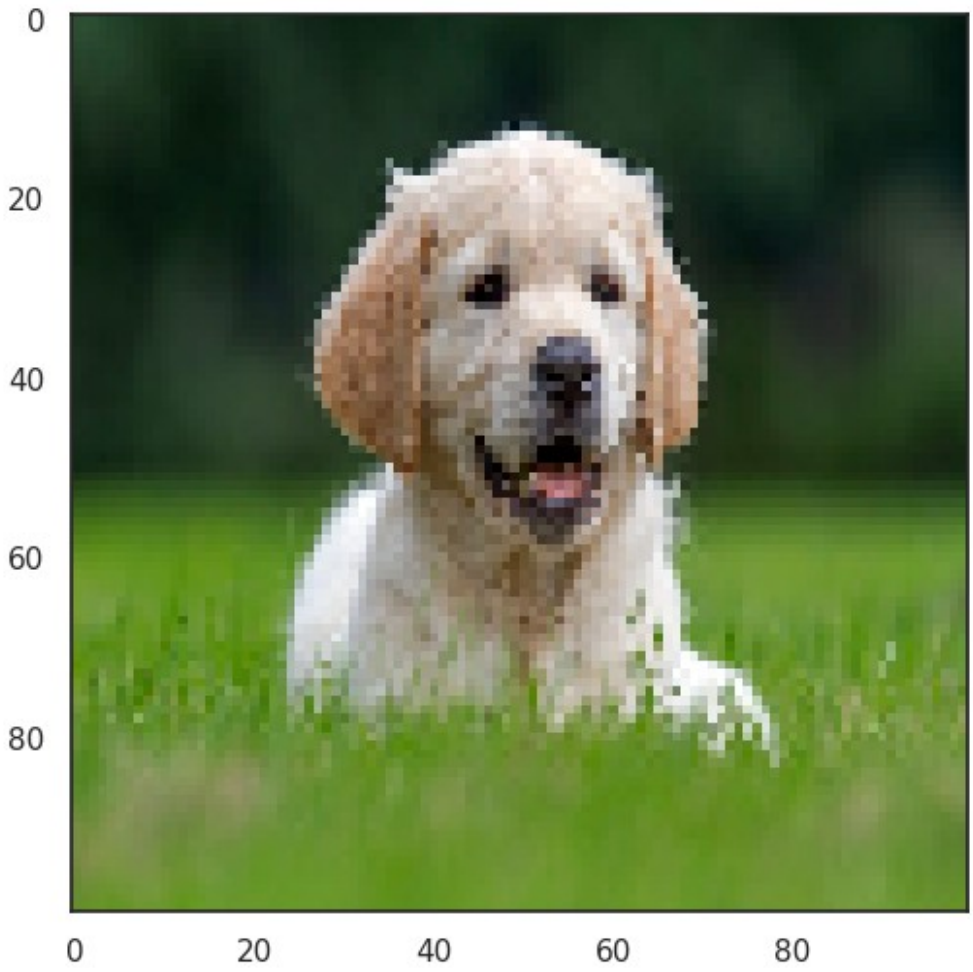
```
<class 'PIL.Image.Image'>
<class 'PIL.Image.Image'>
```

We can plot each image

```
plt.imshow(gray_img,cmap="gray")
plt.show()
```




```
plt.imshow(color_img)
plt.show()
```



Converting an image

We can convert the PIL image to a 3D numpy array of pixel data, and a single image to a batch using the following code:

```
input_arr = tf.keras.preprocessing.image.img_to_array(color_img)


print("image shape",input_arr.shape)
```

```
image shape (100, 100, 3)
```

We need to add the batch dimension before the using Keras, we can add the batch dimension as follows:

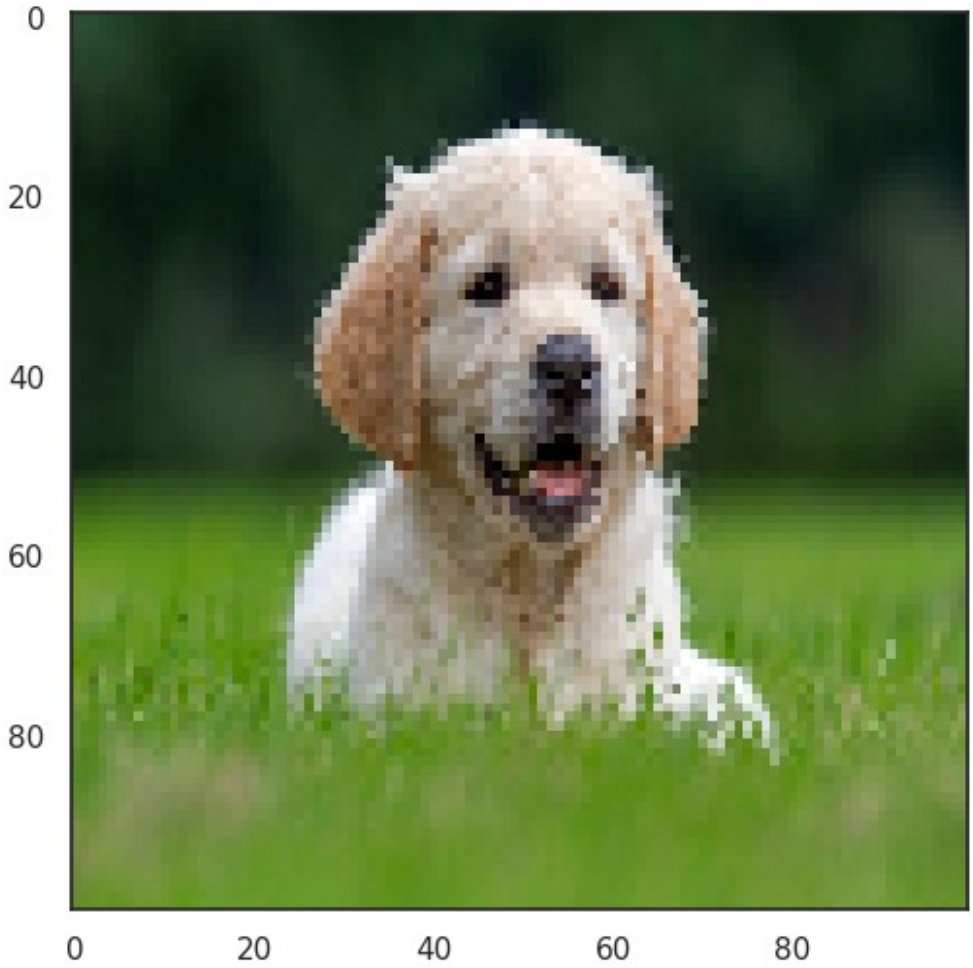
```
# Convert single image to a batch
input_arr_batch = np.array([input_arr])
#or
input_arr_batch=input_arr.reshape(-
1,input_arr.shape[0],input_arr.shape[1],input_arr.shape[2])
```

```
print("image shape plus batch dimension",input_arr_batch.shape)
```

```
image shape plus batch dimension (1, 100, 100, 3)
```

Similarly, we can convert a numpy array of pixel data back to an image.

```
# Convert the numpy array back to an image
color_img = tf.keras.preprocessing.image.array_to_img(input_arr)
plt.imshow(color_img)
plt.show()
```



Saving an image

```
tf.keras.preprocessing.image.save_img('dog_color_img.jpg', color_img)
```

C. Load from directory of images

We can use the `ImageDataGenerator` class in Keras to load train, test, and validation datasets. This is especially useful when working with datasets containing several thousand or

millions of images. More details on content provided in this sub-section can be found in the Tensorflow documentation.

We will first load in our dataset of flowers, and then look at what we have.

The dataset has five types of flowers.

Category	Flower
0	`Daisies`
1	`Dandelions`
2	`Roses`
3	`Sunflowers`
4	`Tulips`

Let's load the data and take a look.

Because the flower dataset is not one of the default small toy datasets available, we'll need to download it using the `keras.utils.get_file()` function.
Note that you will only need to specify the first three arguments.

```
keras.utils.get_file(
    fname=None,
    origin=None,
    untar=False,
    md5_hash=None,
    file_hash=None,
    cache_subdir='datasets',
    hash_algorithm='auto',
    extract=False,
    archive_format='auto',
    cache_dir=None
)
```

```
import pathlib
dataset_url = "https://cf-courses-data.s3.us.cloud-object-
storage.appdomain.cloud/IBMDeveloperSkillsNetwork-ML311-Coursera/
datasets/flower_photos.tgz"

# Download the data and track where it's saved using
tf.keras.utils.get_file in a variable called data_dir
data_dir = keras.utils.get_file(origin=dataset_url,
                                fname='flower_photos',
                                untar=True)
data_dir = pathlib.Path(data_dir)

for folder in data_dir.glob('[!LICENSE]*'):
    print('The', folder.name, 'folder has',
len(list(folder.glob('*.jpg'))), 'pictures')
```

```
image_count = len(list(data_dir.glob('*/*.jpg')))
print(image_count, 'total images')
```

```
Downloading data from https://cf-courses-data.s3.us.cloud-object-
storage.appdomain.cloud/IBMDeveloperSkillsNetwork-ML311-Coursera/
datasets/flower_photos.tgz
228813984/228813984 ━━━━━━━━━━━━━━━━━━━━ 8s 0us/step
The flower_photos folder has 0 pictures
0 total images
```

The files are stored in the following directory.

```
flower_photos
    └── daisy
    └── dandelion
    └── roses
    └── sunflowers
    └── tulips
```

It seems there are a substantial amount of photos for each type of photo.

We can checkout some of the images below.

Visualizing images

```
# Define the correct path to the 'dandelion' folder
dandelion_path = pathlib.Path(data_dir) / "flower_photos" /
"dandelion"

# Get all image files in the folder
dandelion_images = list(dandelion_path.glob('*.jpg'))

# Print the number of images found
print(f"Found {len(dandelion_images)} images in the 'dandelion'
folder.")

# Open the second image safely
if len(dandelion_images) > 1:
    img = PIL.Image.open(dandelion_images[1])  # Open second image
    img.show()
else:
    print("Not enough images in the 'dandelion' folder.")
```

```
Found 898 images in the 'dandelion' folder.
```

```
# Define correct path to the 'roses' folder
roses_path = pathlib.Path(data_dir) / "flower_photos" / "roses"

# Get all image files in the 'roses' folder
roses_images = list(roses_path.glob('*.jpg'))  # Make sure only images
are listed
```

```
# Print the number of images found
print(f"Found {len(roses_images)} images in the 'roses' folder.")

# Open an image safely
if len(roses_images) > 1:
    img = PIL.Image.open(roses_images[1])  # Open the second image
    img.show()
elif len(roses_images) == 1:
    print("Only one image found. Opening the first image.")
    img = PIL.Image.open(roses_images[0])
    img.show()
else:
    print("No images found in the 'roses' folder.")
```

```
Found 641 images in the 'roses' folder.
```

```
# Define the correct path
sunflowers_path = pathlib.Path(data_dir) / "flower_photos" /
"sunflowers"

# Get all image files
sunflowers_images = list(sunflowers_path.glob('*'))  # Get all files

# Print debugging info
print(f"Found {len(sunflowers_images)} files in 'sunflowers' folder.")
print(sunflowers_images)  # Show the actual file paths

# Open an image safely
if len(sunflowers_images) > 1:
    img = PIL.Image.open(sunflowers_images[1])  # Open the second
image
    img.show()
elif len(sunflowers_images) == 1:
    print("Only one image found. Opening the first image.")
    img = PIL.Image.open(sunflowers_images[0])
    img.show()
else:
    print("No images found in 'sunflowers' folder.")

# Define the path
daisy_path = pathlib.Path(data_dir) / "flower_photos" / "daisy"

# Check if the path exists and list contents
if daisy_path.exists():
    print(f"'{daisy_path}' exists.")
    daisy_images = list(daisy_path.glob('*'))  # Get all files, any
format
    print(f"Found {len(daisy_images)} files in 'daisy' folder.")
    print("Files found:", daisy_images)
else:
```

```
    print(f"Path '{daisy_path}' does not exist.")
    daisy_images = []

# Open an image safely
if len(daisy_images) > 1:
    img = PIL.Image.open(daisy_images[1])  # Open the second image
    img.show()
elif len(daisy_images) == 1:
    print("Only one image found. Opening the first image.")
    img = PIL.Image.open(daisy_images[0])
    img.show()
else:
    print("No images found in 'daisy' folder.")
```

We can see that these photos are in various lighting conditions, framing, sizes, and zoom.

```
# The batch size simply specifies the number of images to pass through
our neural network at a time, until the entire training set is passed
through. 32 is the default
batch_size = 32

# Here we set the size of all the images to be 200x200
img_height = 200
img_width = 200
```

Here we split our images into training and validation sets using the `keras.utils.image_dataset_from_directory()` function.

- Because our flower images are sorted into subfolders named after the type of flower, this method can automatically load in images and label them with the correct classes.
- In order to plug these photos into our neural network as an input, we'll have to standardize the image sizes. We can do that with the `image_size` argument.

Let us use `image_dataset_from_directory` to load images off disk:

```
train_ds = tf.keras.utils.image_dataset_from_directory(
  data_dir,
  validation_split=0.2,
  subset="training",
  seed=123,
  image_size=(img_height, img_width),
  batch_size=batch_size)
```

```
Found 3670 files belonging to 1 classes.
Using 2936 files for training.
```

Exercise 3 - Load validation set from the directory

Use the `image_dataset_from_directory` to load the validation set from the directory

```
  validation_ds = tf.keras.utils.image_dataset_from_directory(
  data_dir,
  validation_split=0.2,
  subset="validation",
  seed=123,
  image_size=(img_height, img_width),
  batch_size=batch_size)
```

```
Found 3670 files belonging to 1 classes.
Using 734 files for validation.
```

We can show the class names

```
class_names = train_ds.class_names
class_names
```

```
['flower_photos']
```

Let's look at some sample rows from the dataset we loaded:

```
# .take() will take the first batch from a tensorflow dataset.
# In our case it has taken the first 32 images.
first_batch = train_ds.take(1)

plt.figure(figsize = (25,10))
for img, lbl in first_batch:
    # lets look at the first 10 images
    for i in np.arange(10):
        plt.subplot(2,5,i+1)
        plt.imshow(img[i].numpy().astype('uint8'))
        plt.title(class_names[lbl[i]])
        plt.axis("off")
```

We can see that after importing our images, they're all exactly square with `200 x 200` pixels.

Augmenting Images

Next, we will see how we can use `ImageDataGenerator` class to create tensor image data batches that have been augmented to have certain characteristics. Image data augmentation is useful in expanding the training dataset in order to improve the performance and ability of the model to generalize. Here are some ways in which we can augment images in our dataset:

- **rescale**: rescaling factor
- **rotation_range**: degree of random rotations of the images
- **width_shift_range**: upper bound for random shift, either left or right
- **height_shift_range**: upper bound for random shift, either up or down
- **vertical_flip**: flip the image vertically

Further details on these options can be found in the Tensorflow documentation.

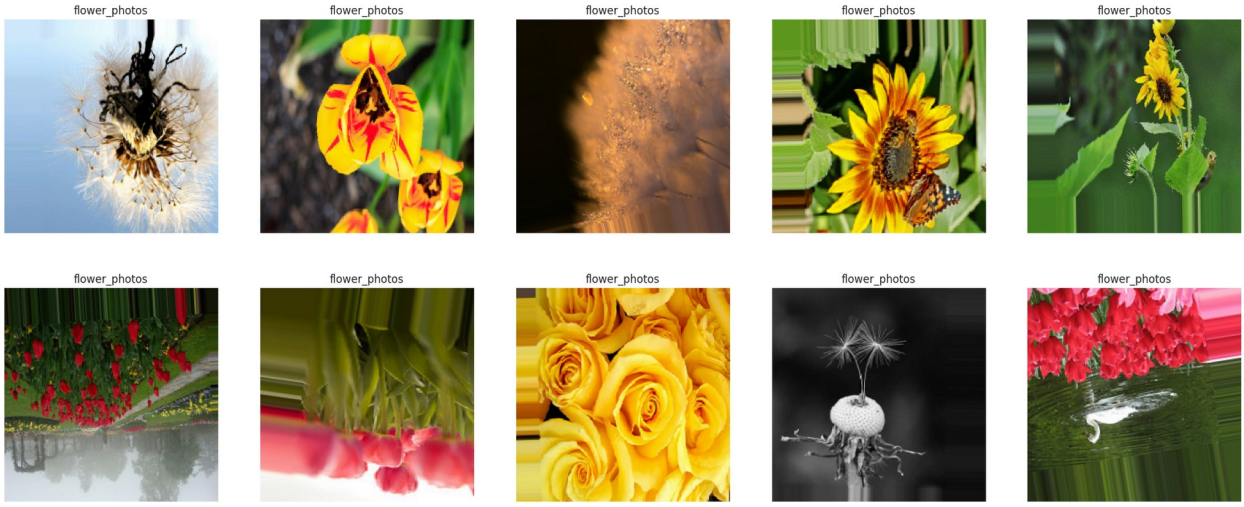
```
import tensorflow as tf

img_gen = tf.keras.preprocessing.image.ImageDataGenerator(
    rescale=1.0 / 255,
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    vertical_flip=True,
)

flowers_data = img_gen.flow_from_directory(data_dir)
images, labels = next(flowers_data)

print(images.shape)
print(labels.shape)
```

```
Found 3670 images belonging to 1 classes.
(32, 256, 256, 3)
(32, 1)
```

Exercise 4 - Visualizing augmented images

Visualize 10 images from the augmented tensor image data batch.

```
plt.figure(figsize=(25, 10))
for i in range(10):
    plt.subplot(2,5,i+1)
    plt.imshow(images[i])
    index = [index for index, each_item in enumerate(labels[i]) if
each_item]
    plt.title(list(flowers_data.class_indices.keys())[index[0]])
    plt.axis("off")
```



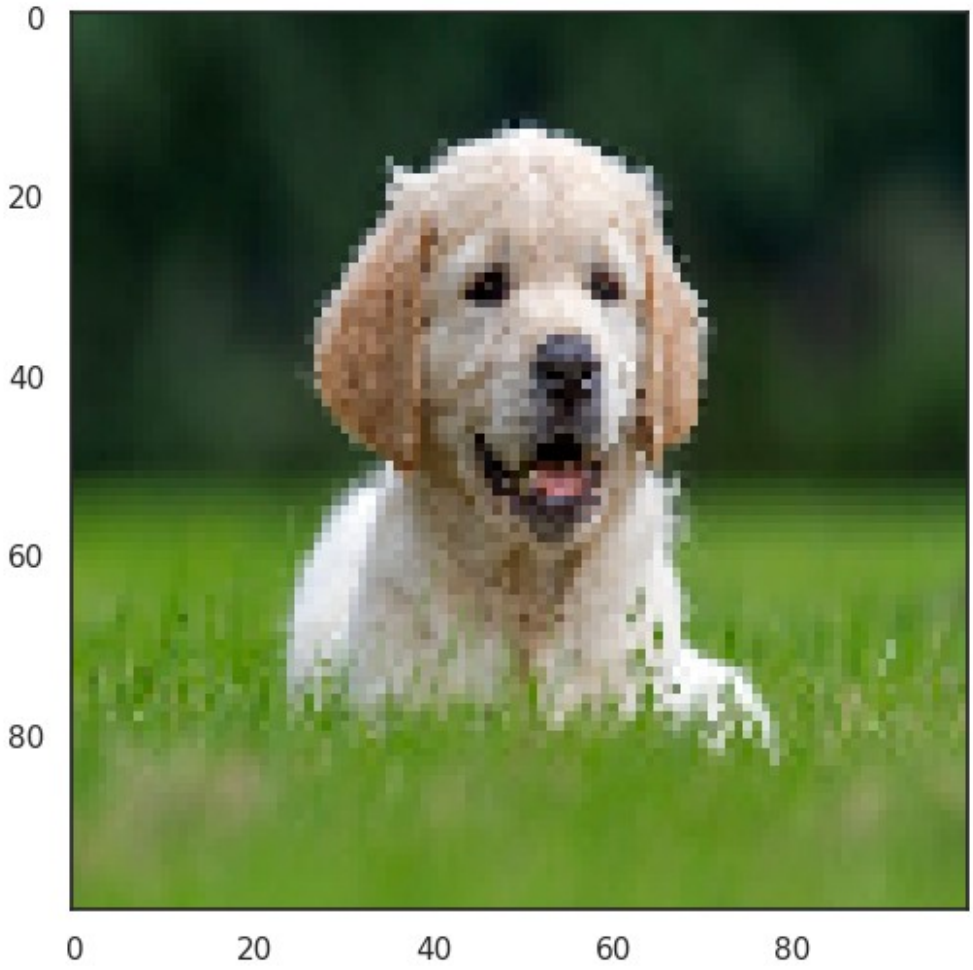
D. Load image from URL

In the *Load individual images as PIL objects section*, we saw how we can download images to our working directory and, using the path of the image we saved, use the `load_img` function to return a PIL Image instance.

We can use custom functions like the one below to load images from a URL. This function was borrowed from statsmaths.

```
def load_image(link, target_size=None):
    import requests
    import shutil
    import os
    _, ext = os.path.splitext(link)
    r = requests.get(link, stream=True)
    with open('temp.' + ext, 'wb') as f:
        r.raw.decode_content = True
        shutil.copyfileobj(r.raw, f)
    img = tf.keras.preprocessing.image.load_img('temp.' + ext,
target_size=target_size)
    return tf.keras.preprocessing.image.img_to_array(img)

URL = 'https://cf-courses-data.s3.us.cloud-object-
storage.appdomain.cloud/IBMDeveloperSkillsNetwork-ML311-Coursera/
labs/Module3/L1/Dog_Breeds.jpg'

img = load_image(URL, target_size=(100, 100))
plt.imshow(img/255)
```

```
<matplotlib.image.AxesImage at 0x795236882290>
```



Authors

Kopal Garg

Kopal Garg is a Masters student in Computer Science at the University of Toronto.

Other Contributors

Richard Ye, Roxanne Li

Change Log

Date (YYYY-MM-DD)	Version	Changed By	Change Description
2022-05-23	0.1	Kopal Garg	Created Lab
2022-05-23	0.1	Richard Ye	Load from directory of images
2022-05-30	0.1	Roxanne Li	Review and edit content
2022-07-19	0.1	Steve Hord	QA pass