

Import necessary libraries

```
import numpy as np
from PIL import Image
import torch
from torchvision import transforms, datasets
from scipy.stats import multivariate_normal
from IPython.display import display
import matplotlib.pyplot as plt

plt.rcParams['figure.figsize'] = (4, 4)
%matplotlib inline
```

Task 1: Simple Image Operations

1. Load image

```
image_path = './data/capybara.jpg'
img = Image.open(image_path)
```

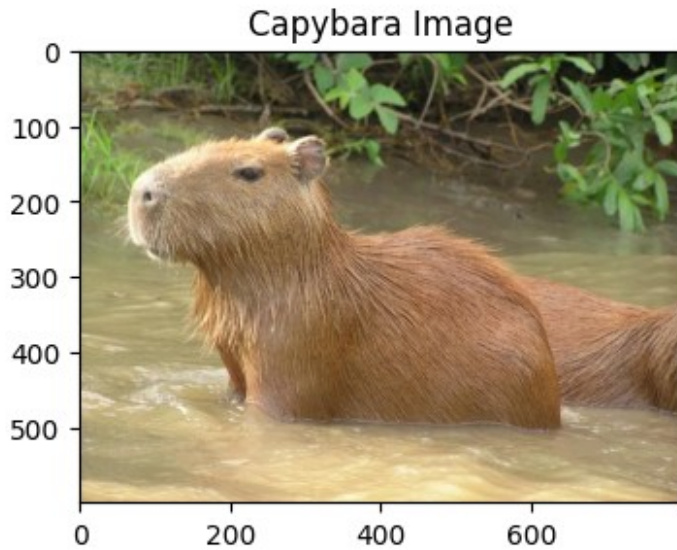
2. Print width, height, and number of channels and plot the image

```
# print the image size
print(f"width: {img.width}, height: {img.height}, channel: {len(img.getbands())}")
```

```
width: 800, height: 600, channel: 3
```

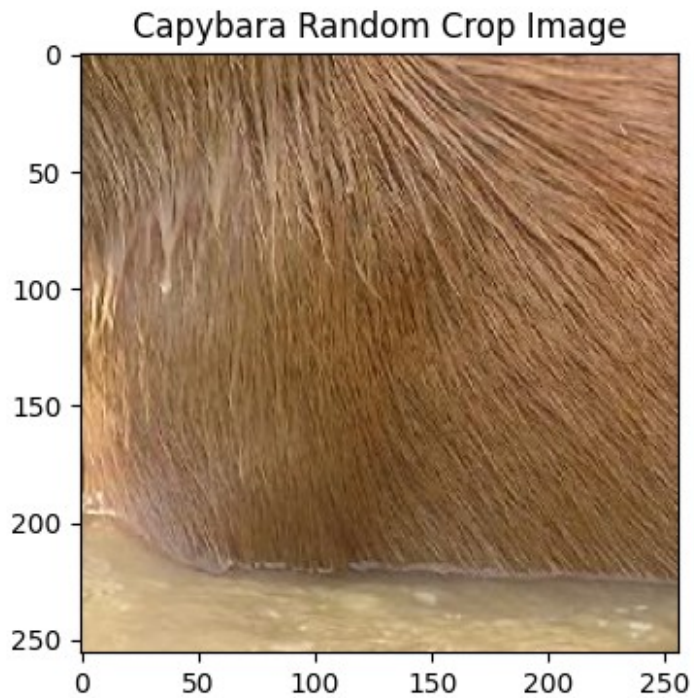
```
# plot the image
plt.title('Capybara Image')
plt.imshow(img)
```

```
<matplotlib.image.AxesImage at 0x310675910>
```



3. Crop a random patch of size 256x256 from the image

```
def get_random_crop(image, crop_height, crop_width):  
    width, height = image.size  
  
    max_x = width - crop_width  
    max_y = height - crop_height  
  
    x = np.random.randint(0, max_x)  
    y = np.random.randint(0, max_y)  
  
    crop = image.crop((x, y, x + crop_width, y + crop_height))  
  
    return crop, x, y  
  
# crop random patch of 256x256  
random_crop_size = 256  
random_crop, random_crop_x, random_crop_y = get_random_crop(img,  
    random_crop_size, random_crop_size)  
  
# plot the cropped image  
plt.title('Capybara Random Crop Image')  
plt.imshow(random_crop)  
  
<matplotlib.image.AxesImage at 0x310633250>
```

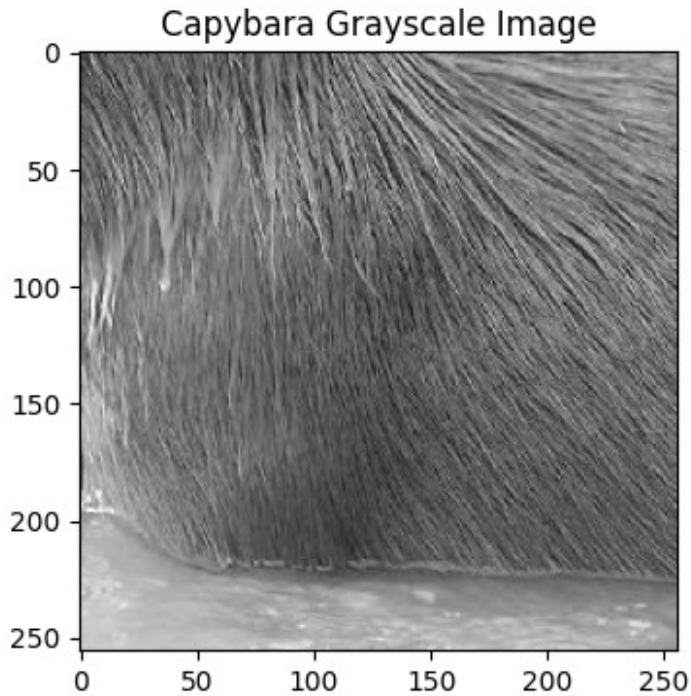


4. Convert the image patch to grayscale

```
# convert to grayscale
random_crop_gray = transforms.Grayscale()(random_crop)

# plot the grayscale image
plt.title('Capybara Grayscale Image')
plt.imshow(random_crop_gray , cmap='gray')

<matplotlib.image.AxesImage at 0x31069e310>
```



5. Insert the grayscale patch back into the original image

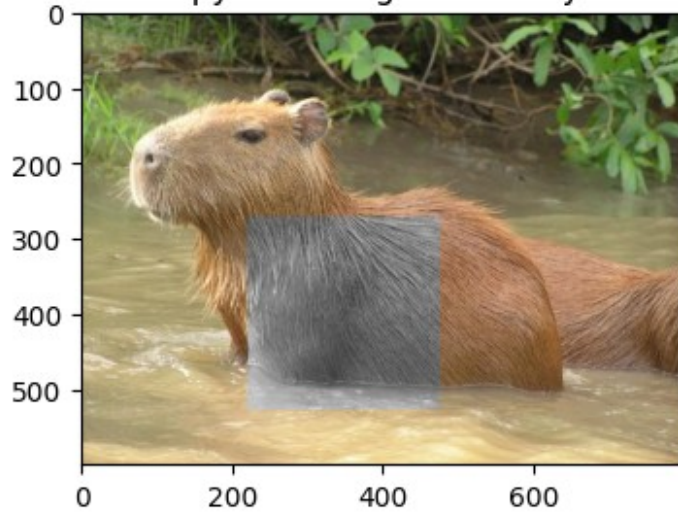
```
# convert to numpy array
img_np = np.array(img)
random_crop_gray_np = np.array(random_crop_gray)

# insert grayscale image back into original image
img_np[random_crop_y:random_crop_y + random_crop_size,
random_crop_x:random_crop_x + random_crop_size] =
np.stack((random_crop_gray_np,) * 3, axis=-1)

# convert the modified numpy array back to a PIL Image
modified_img = Image.fromarray(img_np)

# plot the modified image
plt.title('Modified Capybara Image with Grayscale Patch')
plt.imshow(modified_img)
plt.show()
```

Modified Capybara Image with Grayscale Patch



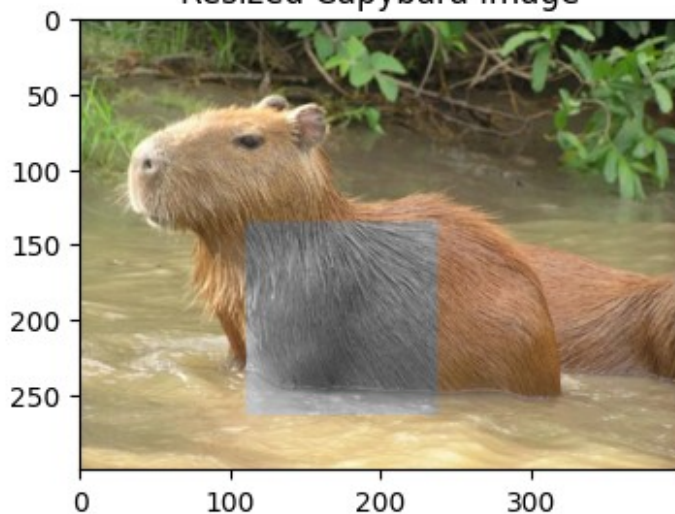
6. Resize the image with the inserted gray patch

```
# resize modified image to 50% of original size
scale = 0.5
resized_img = modified_img.resize((int(modified_img.width * scale),
int(modified_img.height * scale)))

# plot the resized image
plt.title('Resized Capybara Image')
plt.imshow(resized_img)

<matplotlib.image.AxesImage at 0x31060f090>
```

Resized Capybara Image



Task 2: Convolution and Filters

Utils

take care of the range of the value before plot the images

```
def display_img(x, norm=False):
    """
    Displays image
    """
    x = np.copy(x)
    # Normalize if necessary
    if norm:
        x = (x - np.min(x)) / (np.max(x) - np.min(x))
    # Scale and convert to uint8 format for PIL plotting
    x *= 255
    x = x.astype(np.uint8)
    # Handle channel dimension
    if len(x.shape) == 2:
        x = x[:, :, np.newaxis]
    if x.shape[2] == 1:
        x = np.repeat(x, 3, axis=2)

    display(Image.fromarray(x))

def load_img(path, grayscale=False):
    """
    Loads image from given path, converts to float and normalize to
    [0,1] interval
    """
    img = Image.open(path)
    a = np.asarray(img, dtype=np.float32) / 255
    # Convert to grayscale if necessary
    if grayscale:
        a = np.mean(a, axis=2, keepdims=True)
    return a
```

1. Implement convolution operation

```
def conv2d(x: np.ndarray, kernel: np.ndarray) -> np.ndarray:
    """
    Performs 2D convolution operation with a single kernel.

    Args:
        X (array): the input array of shape (H, W, 1).
        kernel (array): the kernel that is convolved over the input.

    Returns:
        out (array): the output array.
```

```

"""
kernel_height, kernel_width = kernel.shape[:2]

output_height = x.shape[0] - kernel_height + 1
output_width = x.shape[1] - kernel_width + 1

result = np.zeros((output_height,
output_width)).astype(np.float32)

for i in range(output_height):
    for j in range(output_width):
        result[i][j] = np.sum(x[i:i + kernel_height, j:j +
kernel_width] * kernel).astype(np.float32)

return result

```

2. Implement a Gaussian filter

```

def gaussian_filter(size, mean=0, var=1) -> np.ndarray:
    """
    Returns an isotropic Gaussian filter with.

    Args:
        size      (int): the size of the kernel.
        mean (float/int): the mean of the Gaussian.
        var  (float/int): the variance of the Gaussian.

    Returns:
        f      (array): the Gaussian filter of shape (size, size,
1).
    """
    x = np.linspace(-size // 2, size // 2, size)
    y = x[:, np.newaxis]

    gaussian = (1 / (np.sqrt(2 * np.pi * var))) * np.exp(-((x ** 2 + y
** 2) / (2 * var)))

    # Normalize the filter
    gaussian /= np.sum(gaussian)

    # Add the mean (if non-zero)
    if mean != 0:
        gaussian += mean

    return gaussian

f = gaussian_filter(size=200, mean=0, var=1000)

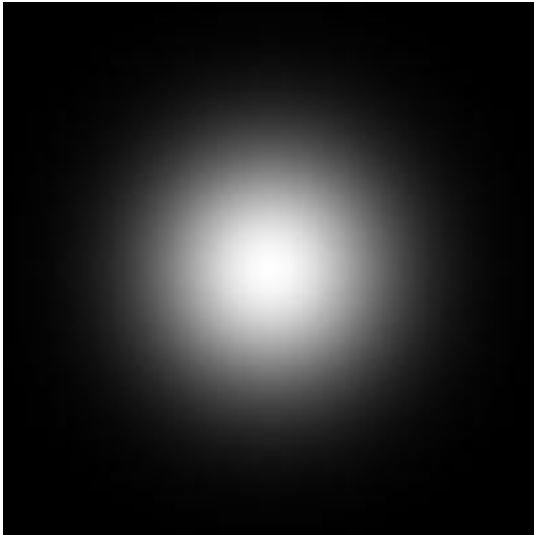
sum = np.sum(f)
print(sum)

```

1.0

```
print('Gaussian filter:')  
display_img(f, norm=True)
```

Gaussian filter:



3. Apply Gaussian filter to image

```
# apply the filter to the image  
img = load_img(image_path, grayscale=True)  
array = np.array(img)
```

```
print('Original image:')  
display_img(array)
```

Original image:



```
filter_size = (200, 200)
mean = 0
variance = 100

# obtain the Gaussian filter
f = gaussian_filter(size=filter_size[0], mean=mean, var=variance)

# normalize the image by dividing 255
array_norm = array / 255

# perform 2D convolution
array_filtered = conv2d(array_norm, f)

# normalize the filtered image by dividing 255
array_filtered /= 255
# scale the image to [0, 255]
array_filtered = (array_filtered - np.min(array_filtered)) /
(np.max(array_filtered) - np.min(array_filtered))

print('Gaussian filtered image:')
display_img(array_filtered)
```

4. Implement and apply Laplacian filter to image

Hint: If you only apply laplacian and use plt to plot without taking care of the value range, you would probably have a hard time to see something

```
# implement the Laplacian filter
def laplace_filter(size: int) -> np.ndarray:
    """
    Returns a Laplacian filter of a specified size.

    Args:
        size (int): the size of the kernel (assumed to be odd).

    Returns:
        f (array): the Laplacian filter of shape (size, size).
    """
    # odd sizes only
    if size % 2 == 0:
        size += 1

    mid = size // 2
    laplacian = np.zeros((size, size))
    laplacian[mid, mid] = -(size**2 - 4)

    for i in range(size):
        for j in range(size):
            if i != mid and j != mid:
                laplacian[i, j] = 1

    return laplacian

# obtain the Laplacian filter
laplacian_filter = laplace_filter(size=3)

print('Original image:')
display_img(array)

Original image:
```



```
# normalize the image by dividing 255
array_norm = array / 255

# apply the Laplacian filter to the image
array_filtered = conv2d(array_norm, laplacian_filter)

# normalize the filtered image by dividing 255
array_filtered /= 255
# scale the image to [0, 255]
array_filtered = (array_filtered - np.min(array_filtered)) /
(np.max(array_filtered) - np.min(array_filtered))

print('Laplacian filtered image:')
display_img(array_filtered)

Laplacian filtered image:
```



5. Problems with handcrafted filters

Hand-crafted filters have limitations, such as limited adaptability, as they are designed for specific features or patterns like edges, corners, or textures, which might not be suitable for the actual patterns in the input data, leading to suboptimal performance. They also require domain expertise and effort to design, which can be time-consuming and costly. Additionally, hand-crafted filters primarily focus on extracting low-level features and may not capture high-level salient features as effectively as learned filters, which are better in learning hierarchical representations of the data.

Task 3: Introduction to PyTorch

1. Convert NumPy array to PyTorch tensor and vice versa

```
# convert the image to a PyTorch tensor
x = torch.from_numpy(array)

print(f"Shape of tensor: {x.shape}")

Shape of tensor: torch.Size([600, 800, 1])
```



```

# swap the axes to match the PyTorch format
x = x.permute(2, 0, 1)

print(f"Shape of tensor: {x.shape}")
Shape of tensor: torch.Size([1, 600, 800])

# swap back the axes to match the original format
x = x.permute(1, 2, 0)

print(f"Shape of tensor: {x.shape}")
Shape of tensor: torch.Size([600, 800, 1])

# convert the tensor back to a numpy array
array = x.numpy()

print(f"Shape of array: {array.shape}")
Shape of array: (600, 800, 1)

```

2. Create a PyTorch convolution operator

```

# create random array for image and kernel
X = np.random.rand(5,5,1)
W = np.random.rand(2,2,1)

print(f"Shape of X: {X.shape}")
print(f"Shape of W: {W.shape}")

Shape of X: (5, 5, 1)
Shape of W: (2, 2, 1)

# convert x,w to PyTorch tensors / swap axes
X_i = torch.from_numpy(X)
X_i = X_i.permute(2,0,1)

W_i = torch.from_numpy(W)
W_i = W_i.permute(2,0,1)

print(f"Shape of tensor: {X_i.shape}")
print(f"Shape of tensor: {W_i.shape}")

Shape of tensor: torch.Size([1, 5, 5])
Shape of tensor: torch.Size([1, 2, 2])

# expand the dimensions of the tensor
X_i = torch.unsqueeze(X_i, dim=0)
W_i = torch.unsqueeze(W_i, dim=0)

print(f"Shape of tensor: {X_i.shape}")
print(f"Shape of tensor: {W_i.shape}")

```

```

Shape of tensor: torch.Size([1, 1, 5, 5])
Shape of tensor: torch.Size([1, 1, 2, 2])

# create a pytorch convolutional operator
conv = torch.nn.Conv2d(in_channels=1, out_channels=1, kernel_size=2,
bias=False)

print(f"Convolutional operator: {conv.weight.shape}")
Convolutional operator: torch.Size([1, 1, 2, 2])

# set the weights of the convolutional operator
conv.weight.data = W_i

# apply the convolutional operator to the image
output = conv(X_i)

print(f"Output shape: {output.shape}")
Output shape: torch.Size([1, 1, 4, 4])

# Remove the last dimension from X and W

print(f"Shape of X: {X.shape}")
print(f"Shape of W: {W.shape}")

Shape of X: (5, 5, 1)
Shape of W: (2, 2, 1)

# apply own convolutional operator to the image
output = conv2d(X, W)

print(f"Output shape: {output.shape}")
Output shape: (4, 5)

```

3. Apply Gaussian filter using nn.Conv2d

```

# permute the axes of the image tensor x
x = x.permute(2, 0, 1)
x = x.unsqueeze(0).float()

# set gaussian filter as the weight of the convolutional operator
gaus_f = torch.from_numpy(f).unsqueeze(0).unsqueeze(0).float()

conv.weight.data = gaus_f

x_img = x / 255 # to calculate gradients

# apply the convolutional operator to the image
xx_filtered = conv(x_img).squeeze().detach().numpy()

```

```
# normalize the filtered image by dividing 255
xx_filtered /= 255
# scale the image to [0, 255]
xx_filtered = (xx_filtered - np.min(xx_filtered)) /
(np.max(xx_filtered) - np.min(xx_filtered))

print('Original image:')
display_img(array)

Original image:
```



```
print('Filtered image:')
display_img(xx_filtered)

Filtered image:
```

