Convolution

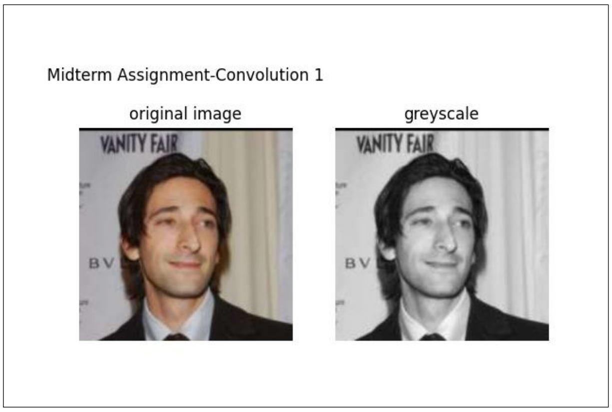
1. convertToGrayscale

Step1: Use the function to convert change the RGB to yCBcr.

Step2: Split the 3 color channels and only return them.

Step3: Open the image and process it by the function.

Step4: Output the original image and the greyscale like what showed on the right.



2. computeDerivativeOfGaussianGradient

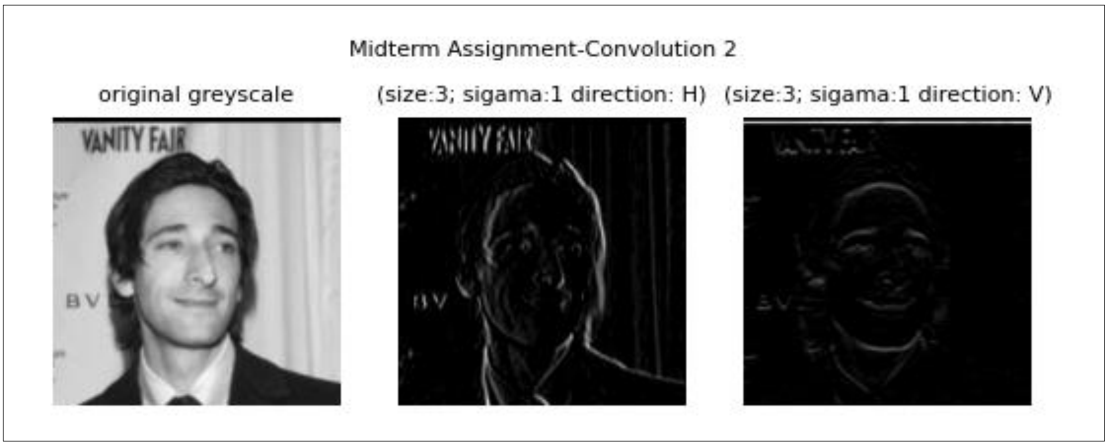
Step1: Use cv2. getGaussianKernel(size=3,3, sigma=1) to get a 1D GaussianKernel.

Step2: Get 2DKernel by calculating the 1D*1D.T

Step3: Convolute the 2D Kernel by horizontal gradient template and vertical gradient template separately.

Step4: Convolute the original image by the two templates got from step 3 and then show the result like below.

```
1d: size: 3, sigma: 1.000000
[[0.27406862]
 [0.45186276]
 [0.27406862]]
2d: size: 3, sigma: 1.000000
[[0.07511361 0.1238414  0.07511361]
 [0.1238414  0.20417996 0.1238414 ]
 [0.07511361 0.1238414  0.07511361]]
[[-0.08033855  0.          0.08033855]
 [-0.0487278   0.          0.0487278 ]
 [-0.0487278   0.          0.0487278 ]]
[[-0.08033855 -0.0487278  -0.0487278 ]
 [ 0.          0.          0.        ]
 [ 0.08033855  0.0487278   0.0487278 ]]
```



PCA

1. LFW

It is can be observed from the LFW dataset that all of the pictures in it are face portraits, and most of them only have a single one and in the middle of the pictures.

2. readImgBatch

Step1: Traversal all the images in the lfw folder, convert them to grayscale by convertTo Grayscale, and rescale them by scaling factor 4 (shrink to 1/4)

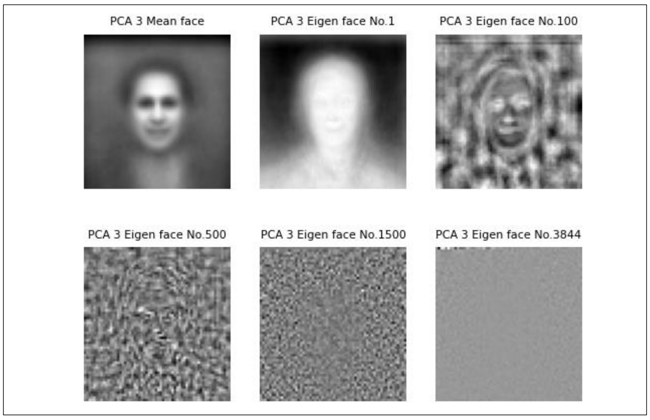
Step2: Put all the pixels which belong to on image in a row, and store all the images in a 2D matrix.

3. computePCA

Step1: Calculate the mean face by np.mean(image_vector, axis=0)

Step2: Subtract the mean face from each feature vector

Step3: Calculate the covariance matrix by np.matmul(subtraction_matrix.T, subtraction_matrix)



Step4: Calculate the eigenvalues and eigenvectors by np.linalg.eigh (np.mat(temp))

Step5: Sort the eigenvectors from large to small

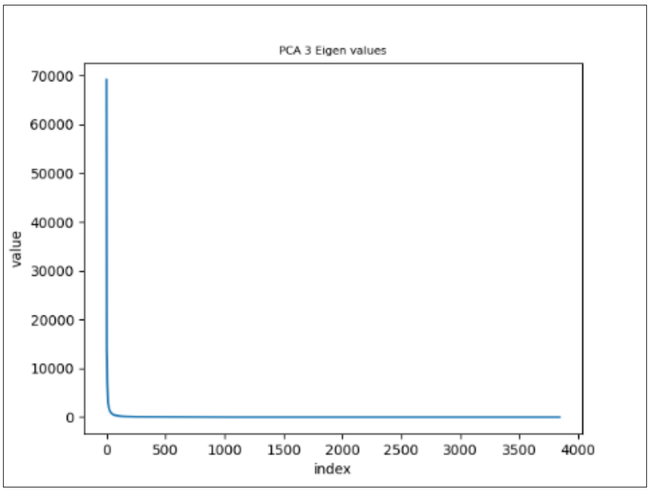
Step6: Output the mean face and three eigenfaces

Note:

- If use the function np. linalg.eig(), the result will be complex numbers.
- Before outputting the eigen-faces, it is necessary to transpose the eigenvectors or output them by columns.

And the result is like the picture above.

A plot of the eigenvalues is shown as right and we have talked about CNN which can be considered the parallel in the lecture as well.



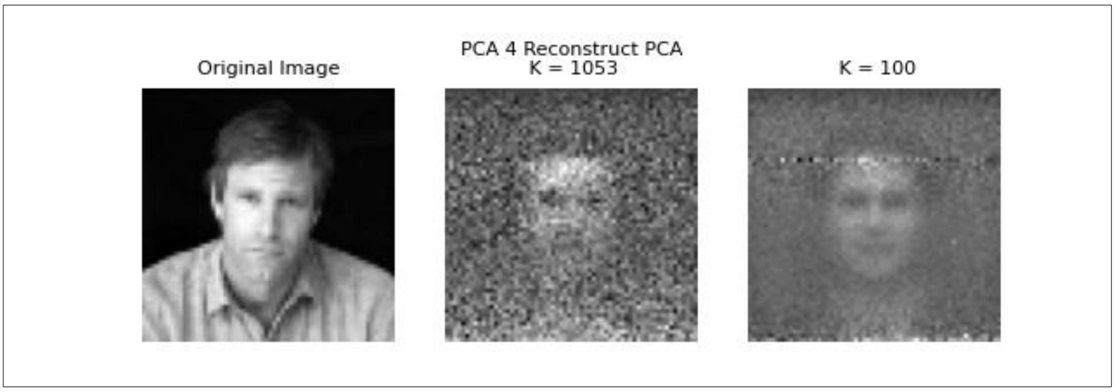
4. reconstructFromPCA

Step1: Traversal eigenvalues and stop by the value which is larger than 0.01 and give the value to k

Step2: Reconstruct the input image from its PCA projection using K principal components.

Step3: Add mean face to the result in Step2

The final result is like below. From the result, it is easy to find that when k=1053, the picture looks more complex than K=100, and the picture which K=100 has already got the main features. So the larger eigenvectors contribute more to the reconstruction.



Appendix

```
import cv2.cv2 as cv2
import os
import matplotlib.pyplot as plt
import numpy as np
import scipy.signal as sgl
from PIL import Image
from skimage.transform import rescale


# convert the colour RGB image to a grayscale luminance image,
# using the luminance as defined in the YCbCr colour space
def convertToGrayscale(rgb_img):
    y_cb_cr_img = rgb_img.convert('YCbCr')
    y, cb, cr = y_cb_cr_img.split()
    return y

# computes the spatial derivatives in the horizontal and vertical
direction
# using convolution by the 2D derivatives of Gaussian filter,
# and apply it to the luminance grayscale image.
def computeDerivativeOfGaussianGradient(img, size, sigma):
    #gaussian = cv2.GaussianBlur(img, (size, size), sigma, sigma)
    # test = [[0.0, 0.0, 0.0], [0.0, 1.0, 0.0,], [0.0, 0.0, 0.0]]
    # Alternative Method: cv2.GaussianBlur(np.array(test), kszie =
(3, 3), sigmaX=1, borderType= cv2.BORDER_CONSTANT)
    kernel_1d = cv2.getGaussianKernel(size, sigma)
    kernel_2d = kernel_1d * kernel_1d.T
    print("1d: size: %d, sigma: %f" % (size, sigma))
    print(kernel_1d)
    print("2d: size: %d, sigma: %f" % (size, sigma))
    print(kernel_2d)
    horizontal_gradient = [[-1, 0, 1], [0, 0, 0], [0, 0, 0]]
    vertical_gradient = [[-1, 0, 0], [0, 0, 0], [1, 0, 0]]
    # convolve the kernel by template
    horizontalTemplate = sgl.convolve2d(kernel_2d, horizontal_gradient,
mode='same',  boundary='wrap', fillvalue=0)
    verticalTemplate = sgl.convolve2d(kernel_2d, vertical_gradient,
mode='same',  boundary='wrap', fillvalue=0)
    # horizontalTemplate = cv2.filter2D(kernel_2d, -1,
horizontalGradient)
    # verticalTemplate = cv2.filter2D(kernel_2d, -1,
verticalGradient)
    print(horizontalTemplate)
```

```
    print(verticalTemplate)
    #horizontalImg = cv2.filter2D(img, -1,
np.array(horizontalGradient))
    #verticalImg = cv2.filter2D(img, -1, np.array(verticalGradient))
    horizontal_img = cv2.filter2D(img, -1, horizontalTemplate)
    vertical_img = cv2.filter2D(img, -1, verticalTemplate)

    plt.subplot(1, 3, 1)
    plt.imshow(img, cmap='gray')
    plt.axis('off')
    plt.title("\noriginal greyscale", size="8")
    plt.subplot(1, 3, 2)
    # cmap='gray' is to make sure output a grayscale not a heat map
    plt.imshow(horizontal_img, cmap='gray')
    plt.axis('off')
    plt.title("Midterm Assignment-Convolution 2\n\n(size:3; sigama:1
direction: H)", size="8")
    plt.subplot(1, 3, 3)
    plt.imshow(vertical_img, cmap='gray')
    plt.axis('off')
    plt.title("\n(size:3; sigama:1 direction: V)", size="8")
    plt.show()


# Create a matrix containing all images by concatenating the vectors.
def readImgBatch(path, endpoint=None):
    container = []
    for root, dirs, files in os.walk(path):
        for file in files:
            path = os.path.join(root, file)
            input_img = Image.open(path)
            # convert the input image to grayscale and scale down it to
1/4 of the original size.
            scaling_img =
rescale(np.array(convertToGrayscale(input_img)), 0.25,
anti_aliasing=False)
            # reshape(-1)
            container.append(scaling_img.ravel())
    return container


# computes the Principal Component Analysis
def computePCA(image_vector):
    average_image = np.mean(image_vector, axis=0)
```

```
    # calculate the subtraction matrix
    subtraction_matrix = image_vector - average_image
    temp = np.matmul(subtraction_matrix.T, subtraction_matrix)
    eigen_values, eigen_vectors = np.linalg.eigh(np.mat(temp))
    index = np.argsort(eigen_values)[::-1]
    eigen_values = eigen_values[index]
    eigen_vectors = eigen_vectors[:, index]
    return average_image, eigen_values, eigen_vectors


# calculate index of K by compare with target
def calculateIndexOfK(eigenvalues, target):
    for i in range(len(eigenvalues)):
        if eigenvalues[i] <= target:
            return i

# reconstruct the input image from its PCA projection using K
# principal components.
def reconstructFromPCA(k, eigenvectors, image, mean):
    low_image = np.matmul(image, eigenvectors[:, 0:k])
    reconstruct_image = np.matmul(low_image, eigenvectors[:, 0:k].T) +
mean
    return reconstruct_image


# Display colour and grey image side by side.
img = Image.open('Adrien_Brody_0008.jpg')
grayImg = convertToGrayscale(img)
plt.subplot(1, 2, 1)
plt.imshow(img)
plt.axis('off')
plt.title("Midterm Assignment-Convolution 1\n\noriginal image")
plt.subplot(1, 2, 2)
# cmap='gray' is to make sure output a grayscale not a heat map
plt.imshow(grayImg, cmap='gray')
plt.axis('off')
plt.title("\n\ngreyscale")
plt.show()


# Display the 2D horizontal and vertical derivatives of Gaussian
kernels
# and the resulting gradient images as grayscale images.
computeDerivativeOfGaussianGradient(np.asarray(grayImg), 3, 1)
```

```
# PCA 2-3
imgCollection = readImgBatch("lfw-a")
averageImage, eigenValues, eigenVectors = computePCA(imgCollection)
img_height, img_width = rescale(np.array(grayImg), 0.25,
anti_aliasing=False).shape

eigenVectors = eigenVectors.T
pcaImg1 = eigenVectors[0].reshape((img_height, img_width))
pcaImg100 = eigenVectors[99].reshape((img_height, img_width))
pcaImg500 = eigenVectors[499].reshape((img_height, img_width))
pcaImg1500 = eigenVectors[1499].reshape((img_height, img_width))
pcaImg3844 = eigenVectors[3843].reshape((img_height, img_width))

plt.subplot(2, 3, 1)
plt.imshow(averageImage.reshape((img_height, img_width)), cmap="gray")
plt.axis('off')
plt.title("PCA 3 Mean face", size="8")
plt.subplot(2, 3, 2)
plt.imshow(pcaImg1, cmap="gray")
plt.axis('off')
plt.title("PCA 3 Eigen face No.1", size="8")
plt.subplot(2, 3, 3)
plt.imshow(pcaImg100, cmap="gray")
plt.axis('off')
plt.title("PCA 3 Eigen face No.100", size="8")
plt.subplot(2, 3, 4)
plt.imshow(pcaImg500, cmap="gray")
plt.axis('off')
plt.title("PCA 3 Eigen face No.500", size="8")
plt.subplot(2, 3, 5)
plt.imshow(pcaImg1500, cmap="gray")
plt.axis('off')
plt.title("PCA 3 Eigen face No.1500", size="8")
plt.subplot(2, 3, 6)
plt.imshow(pcaImg3844, cmap="gray")
plt.axis('off')
plt.title("PCA 3 Eigen face No.3844", size="8")
plt.show()

#  plot eigenValues
X = []
for i in range(3844):
    X.append(i)
plt.plot(X, eigenValues)
```

```
plt.title("PCA 3 Eigen values", size="8")
plt.xlabel("index")
plt.ylabel("value")
plt.show()

# PCA-4
k = calculateIndexOfK(eigenValues, 0.01)
reconstructImage1 = reconstructFromPCA(k, eigenVectors,
imgCollection[0], averageImage)
reconstructImage2 = reconstructFromPCA(100, eigenVectors,
imgCollection[0], averageImage)
plt.subplot(1, 3, 1)
plt.imshow(imgCollection[0].reshape((img_height, img_width)),
cmap="gray")
plt.axis('off')
plt.title("\nOriginal Image", size="8")
plt.subplot(1, 3, 2)
plt.imshow(reconstructImage1.reshape((img_height, img_width)),
cmap="gray")
plt.axis('off')
plt.title("PCA 4 Reconstruct PCA\n K = " + str(k), size="8")
plt.subplot(1, 3, 3)
plt.imshow(reconstructImage2.reshape((img_height, img_width)),
cmap="gray")
plt.axis('off')
plt.title("\n K = 100", size="8")
plt.show()
```