سید محمد طاها طباطبایی – تمرین سری پنجم

9812762838

چکیده:

در تمرین اول، هرم گوسی و لاپلاسی را ساختیم و نتیجه حاصل رویت شد. تمرین دوم، تاثیر خاصیت جدایی‌پذیری، که همان قابلیت اعمال فیلتر دوبعدی با استفاده از فیلتر های یک‌بعدی است و خاصیت آبشاری که همان کوچکتر شدن متوالی رزولوشن تصاویر هرم بعد از اعمال فیلتر بلرکننده که باعث می‌شود کیفیت تصویر تغییر نکند است. بخش سوم، به بررسی این نکته می‌پردازد که در ازای بار محاسباتی بیشتر برای ایجاد هرم، چه مزیت و کاربردی برای ما ایجاد می‌شود. بخش چهارم، مقایسه استفاده از فیلتر میانگین گیر به جای گوسی، و نحوه تاثیر آن در تصاویر حاصل از هرم است. تمرین پنجم، به پیاده‌سازی و بررسی تبدیل ویولت می‌پردازد که در جهات مختلف، یک فیلتر بالاگذر بر تصویر اعمال می‌کند. در تمرین ششم نیز، متد ویولت و هرم گوسی با یکدیگر مقایسه شدند تا بررسی شود نتیجه حاصل از کدام روش بهتر است و آیا مزیتی در استفاده از هرم گوسی به جای ویولت وجود دارد.

توضیحات فنی:

* توضیح جزییات پیاده‌سازی توابع در محل پیاده‌سازی کد، به صورت کامنت و داکیومنت نوشته شده است.

5.1.1

در سلول اول، هرم گوسی و در سلول بعدی، با کمک هرم گوسی ساخته شده، هرم لاپلاسی را تشکیل دادیم. برای ساخت هرم گوسی از تابع gasussina\_pyramid استفاده کردم. در این تابع، یک فیلتر گوسی باینامینال با عرض 5 با ضرایب به شکلی که در کد مشخص است، ساخته می‌شود و با کمک تابع convolve2d ، در تصویر که در هر مرحله ابتدا پدینگ نیز به آن اضافه شده، کانوالو می‌شود. در نهایت تصویر حاصل با برداشتن سطر و ستون های آن به صورت یکی در میان(گام 2 حرکت در اندیس دهی) به لیست مربوط به نگهداری هرم، افزوده می‌شود.

تابع Laplacian\_pyramid برای ساخت هرم لاپلاسی، از روی هرم گوسی داده شده به ورودی استفاده می‌شود. در این تابع، ابتدا آخرین تصویر در هرم گوسی، به عنوان اولین تصویر هرم لاپلاسی قرار داده می‌شود، تا از روی آن بقیه تصاویر هرم لاپلاسی ساخته شود. در حلقه، هر باز، با شروع از اندیس انتهایی، یک تصویر از هرم گوسی به ابعاد سطح پایین تر خود می‌رسد، سپس از تفاصل تصویر گوسی به‌دست آمده و تصویر اندیس قبلی گوسی، برای ساخت تصویر لاپلاسی جدید استفاده می‌کنیم.

5.1.2

خاصیت جدایی‌پذیری از این جهت مفید است که با کمک این ویژگی، می‌توان تصاویر(سیگنال های دو بعدی) را که نیازمند فیلترهای دوبعدی برای پردازش هستند، به کمک این ویژگی که فیلتر دو بعدی قابلیت جداسازی به فیلتر های یک بعدی دارد، با کمک فیلتر های یک بعدی پیاده‌سازی و پردازش کرد.

خاصیت آبشاری بودن هرم، به ما کمک می‌کند تا به جای اعمال یک فیلتر گوسین با سایز بالا(یا کاهش ابعاد تصویر)، با افزایش سیگما، نتایج مشابه ولی با سرعت بهتر کسب کنیم. می‌دانیم که در حالت عادی، پس از یکبار اعمال فیلتر گوسین، باید ابعاد تصویر را کاهش دهیم(یا افزایش سایز پنجره). خاصیت آبشاری بیان می‌کندکه می‌توان همین اثر را با افزایش سیگما و بدون نیاز به افزایش سایز پنجره یا کاهش ابعاد تصویر با بار پردازشی کمتر، به دست آورد.

برای بهبود سرعت، این ایده مطرح می‌شود که به جای استفاده از ϭ ثابت، از ϭ متغییر در هر گام، استفاده کنیم. ایده ای مشابه با عملکرد الگوریتم SIFT که در آن، فقط پس از چند گام اعمال فیلتر گوسی با سیگما های متفاوت، ابعاد تصویر را کاهش می‌دادیم.

سودوکد:

function GaussianPyramid(image, sigma):

# Initialize pyramid list

pyramid = [image]

# Loop for each step

for i in range(3):

# Apply Gaussian filter with sigma

filtered = GaussianFilter(pyramid[i], sigma)

# Add the filtered image to the pyramid

pyramid.append(filtered)

# Update sigma value

sigma \*= sqrt(2)

return pyramid

function GaussianFilter(image, sigma):

# Apply Gaussian filter with given sigma in x-dim and y-dim

...

5.1.3

What is the maximum number of levels you can have in an approximation pyramid representation?

حداکثر تعداد سطوح ممکن، به شکل زیر محاسبه می‌شود:

\* سطح صفر را تصویر با طول اصلی و سطح j را، تصویر با طول 1×1 فرض می‌کنیم.(برای مثال پیاده‌سازی شده در کد، 9 سطح داریم، که تصویر با اندیس 8، دارای ابعاد 1×1 است. اگر تصویر اولیه را در شمارش محسوب نکنیم، j سطح خواهیم داشت نه j+1.

اگر به تصاویر ساخته شده در خروجی کد توجه کنیم، می‌بینیم که در تصویر با اندیس 9، دیگر تفاوتی نسبت به مرحله قبل که تصویر با اندیس 8 و ابعاد 1×1 است ندارد، یعنی هرم بیشتر از این تقسیم نمی‌شود.

What is the total number of pixels in the pyramid?

تعداد کل پیکسل ها، برابر مجموع پیکسل های هر سطح است. فرض کنیم در سطح k، یک تصویر با ابعاد داریم، آنگاه:

که در سطح صفر، برابر ابعاد تصویر اصلی، و در سطح آخر برابر 1 است.

How does this number compare with the original number of pixels in the image? Since this number is larger than the original pixel number, what are some of the benefits of using the approximation pyramid? (give some examples)

تعداد پیکسل های تولید شده در این روش بیشتر است.

* لبه‌یابی
  + در سطوح بالاتر تخمینی از لبه های جسم به دست می‌آوریم، و در سطوح پایین‌تر، فقط در ناحیه اطراف که تخمین زده ایم، به لبه‌یابی می‌پردازیم.
* یافتن یک تمپلیت در تصویر
  + ابتدا تصویر و تمپلیت را به سطوح بالا می‌بریم. در لول بالاتر با یک ترشولد مناسب، یک تخمین از محل های وجود تمپلیت به‌دست می‌آوریم. سپس به سطح پایین‌تر برمی‌گردیم و محدوده تخمین را دقیق‌تر می‌کنیم. تا جایی ادامه می‌دهیم که بتوانیم تمپلیت را در تصویر پیدا کنیم.
* فشرده سازی
  + با چشم پوشی از کمی از دست رفتن دیتا، می‌توان تصویر را در هر سطح فشرده کرد و مطابق نیاز از تصویر فشرده شده در لول خاصی به جای کل تصویر استفاده کرد.

Repeat the step for the prediction residual pyramid

با توجه به اینکه ابعاد هرم prediction residual مشابه هرم approximation است، محاسبات انجام شده برای این هرم نیز صادق است. البته می‌توان این نکته را در نظر گرفت که اگر به جای استفاده از ماتریس هایی با ابعاد مشابه ماتریس های هرم قبل، از ماتریس های sparse استفاده کنیم، می‌توان حجم اطلاعات ذخیره‌سازی را کاهش داد و فشرده‌سازی بهتری داشته باشیم، اما در حالت کلی، در این هرم نیز تعداد پیکسل های تولید شده بیشتر از تصویر اصلی است. در این هرم، تفاوت اصلی این است که در هر سطح، به جای نگهداری کلیات تصویر، جزئیات نگهداری می‌شوند.

5.1.4

برای پیاده سازی این بخش، دو تابع pyramid\_reconstruct و approximation\_pyramid را تعریف کردیم. تابع approximation، مشابه محاسبه هرم گوسی، اما با کرنل فیلتر میانگین گیر، روی تصویر اصلی اعمال می‌شود. در این تابع، متد مورد استفاده برای کاهش ابعاد تصویر نیز، طبق خواسته صورت سوال، replication یا همان nearest neighbor است.

5.1.5

در این بخش، تبدیل ویولت و معکوس آن توسط توابع لایبرری pythonWavlet محاسبه شد. در ادامه، معیار های MSE و PSNR برای تصویر حاصل از این روش محاسبه و با تصویر حاصل از تمرین قبلی خواسته شده، مقایسه شد. نتایج مقایسه، مطابق جدول زیر است. دلیل اینکه در روش ویولت، تصویر خروجی و تصویر اصلی دقیقا یکسان نیست، این است که در این روش، از فیلتر های بالاگذری استفاده می‌کنیم که مقداری از جزئیات تصویر را در هر مرحله نادیده می‌گیرد.

|  |  |  |
| --- | --- | --- |
|  | **Gaussian** | **Wavelet** |
| **MSE** | 0.000000 | 0.001003 |
| **PSNR** | infinite | 78.116645 |

5.1.6

در این تمرین، مطابق فرمول داده شده، یک تابع به نام coefficientQuantizer پیاده‌سازی شده است، که مطابق فرمول، ضرایب را گسسته‌سازی می‌کند. سپس از ضرایب حاصل از خروجی این تابع، برای ساخت تصویر در تبدیل معکوس ویولت استفاده می‌شود. علاوه بر خواسته صورت سوال که خواسته شده بود، برای ضریب گسسته‌سازی =2 تصویر را بازسازی کنیم، برای مقایسه بهتر، ضریب 50 و 255 نیز تست شد. نتایج حاصل به شکل زیر است.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **ℽ = 2** | **ℽ = 50** | **ℽ = 255** |
| MSE | 0.962563 | 255.461372 | 4909.622864 |
| PSNR | 48.296513 | 24.057551 | 11.220322 |

بررسی مقادیر، نشان می‌دهد که هر چقدر ضرایب را با شدت بیشتری گسسته‌سازی کنیم، عکس بازسازی شده تفاوت بیشتری با عکس اولیه پیدا می‌کند. طبیعتا برای مقدار ضریب 255، تصویر حاصل سیاه و سفید شد. به نظر می‌رسد، در ازای سرعت بیشتر ویولت در محاسبات، اما دقت آن از هرم گوسی پایین تر است.

import numpy as np

import cv2

import math

from sklearn.metrics import mean\_squared\_error

from skimage.metrics import peak\_signal\_noise\_ratio

from scipy.signal import convolve2d

import pywt

def gaussian\_pyramid(image, n\_levels=6):

    """

    Compute the Gaussian pyramid

    Inputs:

        - image: Input image of size (N,M)

        - n\_levels: Number of stages for the Gaussian pyramid

    Returns:

        Desired gaussian pyramid

    """

    # approximate length 5 Gaussian filter using binomial filter

    a = 0.4

    b = 1./4

    c = 1./4 - a/2

    filt =  np.array([[c, b, a, b, c]])

    pyr = [image]

    for i in np.arange(n\_levels):

        # zero pad the previous image for convolution

        # boarder of 2 since filter is of length 5

        p\_0 = np.pad( pyr[-1], (2,), mode='constant' )

        # convolve in the x and y directions to construct p\_1

        p\_1 = convolve2d( p\_0, filt, 'valid' )

        p\_1 = convolve2d( p\_1, filt.T, 'valid' )

        # DoG approximation of LoG

        pyr.append( p\_1[::2,::2] )

    return pyr

def laplacian\_pyramid(gaussian\_pyr):

    """

    Compute the laplacian pyramid

    Inputs:

        - gaussian\_pyr: Input gaussian pyramid

    Returns:

        Desired laplacian pyramid

    """

    # details pyramid(laplacian)

    # first index should be, the last level of gaussian pyramid

    # alogorithm bulid the pyramid by scaling up the last gaussian pyramid element

    laplacian\_top = gaussian\_pyr[-1]

    n\_levels = len(gaussian\_pyr) - 1

    laplacian\_pyr = [laplacian\_top]

    for i in range(n\_levels,0,-1):

        # this is the size of gaussian level to be expanded

        # it shuold be equal to index before itself

        size = (gaussian\_pyr[i - 1].shape[1], gaussian\_pyr[i - 1].shape[0])

        # openCV pyrUp make the pyramid transform

        gaussian\_expanded = cv2.pyrUp(gaussian\_pyr[i], dstsize=size)

        # subtraction operation

        laplacian = np.subtract(gaussian\_pyr[i-1], gaussian\_expanded)

        laplacian\_pyr.append(laplacian)

    return laplacian\_pyr

def pyramid\_reconstruct(gaussian\_pyr):

    """

    Reconstruct the original image, using provided gaussian (or approximation) pyramid

    NOTE: its really simillar to 'laplacian\_pyramid' function, just has one more step;

    that is the summation step, where we add laplacian image and expanded gaussian

    in order to build next level image in the pyramid

    Inputs:

        - gaussian\_pyr: Input gaussian pyramid

    Returns:

        Desired laplacian pyramid and reconstructed pyramid

    """

    laplacian\_top = gaussian\_pyr[-1]

    n\_levels = len(gaussian\_pyr) - 1

    laplacian\_pyr = [laplacian\_top]

    reconstruct\_pyr = [laplacian\_top]

    for i in range(n\_levels,0,-1):

        size = (gaussian\_pyr[i - 1].shape[1], gaussian\_pyr[i - 1].shape[0])

        gaussian\_expanded = cv2.pyrUp(gaussian\_pyr[i], dstsize=size)

        laplacian = np.subtract(gaussian\_pyr[i-1], gaussian\_expanded)

        laplacian\_pyr.append(laplacian)

        # laplacian and expanded gaussian, make next level iamge in reconstruction pyramid

        reconstruct = np.add(laplacian , gaussian\_expanded)

        reconstruct\_pyr.append(reconstruct)

    return laplacian\_pyr , reconstruct\_pyr

def box\_filter(image,windowSize=3,imagePaddingSize=0):

    """

    Apply averaging filter on Input image. Convolve kernel with size (windoSize,windowSize).

    Inputs:

        - image: Input image of size (N,M)

        - windowSize: Size of averaging kernel

        - imagePaddingSize: Size of image padding. assumed to be eqaul in length and width

    Returns:

        Smoothed image with (windowSize\*windowSize) averaging kernel

    """

    width = image.shape[0]

    length = image.shape[1]

    size = windowSize-1

    # if input image has padding, we drop the padding.

    # using 'uint8' to have pixels in range (0,255).

    newImage = np.zeros((width-(2\*imagePaddingSize),length-(2\*imagePaddingSize)),dtype='uint8')

    for i in range(0,length,1):

        for j in range(0,width,1):

            # calculate proper boundaries for window

            # in left and top edges, indexes should be greater than 0

            start = (max(j-size, 0),max(i-size, 0))

            # in right and down edges, indexes should be less than image length and width

            end = (min(j+size, width-1),min(i+size, length-1))

            # crop a part of image which fits to kernel window

            buffer = image.copy()[start[0]:(end[0]+1), start[1]:(end[1]+1)][0]

            # averaging and replacing in the output image

            buffer\_mean = np.mean(buffer)

            newImage[j,i] = buffer\_mean

    return newImage

def replication(image,zoom\_factor=0.5):

    '''

    The nearest neighbor alogorithm, which is the simplest way to interpolate.

    also, shrink image into size, quarter than the input image size.

    Inputs:

        - image: Inpute image of size (N,M)

        - zoom\_factor: Shrinking ratio

    Returns:

        Shirinked image

    '''

    # output image size calculation

    newWidth = math.floor(image.shape[1]\*zoom\_factor)

    newLength = math.floor(image.shape[0]\*zoom\_factor)

    # new image using nearest neighbor method

    return cv2.resize(image ,(newWidth,newLength), interpolation=cv2.INTER\_NEAREST)

def approximation\_pyramid(image , n\_levels):

    """

    Compute the approximation pyramid (gaussian pyramid, but using

    averaging kernel instead of gaussian kernel)

    Inputs:

        - image: Input image of size (N,M)

        - n\_levels: Number of stages for the approximation pyramid

    Returns:

        Desired approximation pyramid

    """

    # first level of pyramid is the original iamge

    level\_zero = image.copy()

    approxi\_pyr = [level\_zero]

    for \_ in range(n\_levels):

        # applying averaging filter

        level\_zero = box\_filter(level\_zero,windowSize=2)

        # applying replication for interpolation

        level\_zero = replication(level\_zero)

        approxi\_pyr.append(np.asarray(level\_zero))

    return approxi\_pyr

def mean\_square\_error(imageSource, imagetarget):

    """

    The "Mean Squared Error" between the two images is the

    sum of the squared difference between the two images.

    the lower the error, the more "similar" the two images are.

    NOTE: the two images must have the same dimension

    Inputs:

        - imageSource: The source image, we want to calculate the target image difference of

        - imageTarget: The target image, we calculate how far it is from the source

    Returns:

        The MSE

    """

    # cumulative difference

    err = np.sum((imageSource.astype("float") - imagetarget.astype("float")) \*\* 2)

    # divide by length\*width

    err /= float(imageSource.shape[0] \* imageSource.shape[1])

    return format(err,'.6f')

def coefficientQuantizer(coeff,step=2):

    """

    Quantize the wavelet coeddicients, using the formula in the exercise description.

    The formula, simply divide coefficients by given 'step', and round them, and again

    multiply by 'step'. in this way, we have quantized coefficients.

    Inputs:

        - coeff: Given coefficients to be quantized

        - step: Scale of quantization

    Returns:

        New coefficients

    """

    coeff\_new = step \* np.sign(coeff) \* np.floor(np.abs(coeff)/step)

    return coeff\_new

def PSNR(srcImage,testImage):

    """

    Implementation of 'Peak Signal to Noise Ratio' method

    using, sci-kit image library.

    The greater the result, the more "similar" the two images are.

    Inputs:

        - srcImage: The source image, we want to calculate the target image difference of

        - testImage: The target image, we calculate how similar it is with the source

    Returns:

        The PSNR

    """

    return peak\_signal\_noise\_ratio(srcImage,testImage)

def wavelet\_payramid(image,n\_levels,normalization=False):

    """

    Computing wavelet pyramid, using haar method.

    NOTE: if you normalize the coefficients (by setting 'normalization=False') you

    can not rebnild the original image properly. In order to rebuild the original image you

    need to transform the coefficients back to the original domain.

    Inputs:

        - image: Provided image we want to decomposite.

        - n\_levels: Number of stages for wavelet pyramid

        - normalization: A flag, which consider normalization over coefficients

    Returns:

        Desired wavelet pyramid in form of array, Desired wavelet pyramid in form of list

    """

    coeffs = pywt.wavedec2(image, 'haar', mode='periodization', level=n\_levels,)

    if normalization:

        coeffs = normalize(coeffs,1,0)

    c\_matrix, c\_slices = pywt.coeffs\_to\_array(coeffs)

    return c\_matrix , coeffs

def normalize(array,newMax,newMin):

    """

    A simple normalization function.

    Inputs:

        - array: Array to be normalized

        - newMax: Max of new range.

        - newMin: Min of new range.

    Returns:

        Normalized array.

    """

    if isinstance(array, list):

        return list(map(normalize, array,newMax,newMin))

    if isinstance(array, tuple):

        return tuple(normalize(list(array),newMax,newMin))

    normalizedData = (array-np.min(array))/(np.max(array)-np.min(array))\*(newMax-newMin) + newMin

    return normalizedData