

Hybrid image illusion using convolutional filtering

Introduction

Human eye's perception depends strongly on the distance between the object and the observer. If the distance was small, the eye start focusing more on the sharp details of the image (high frequency), however at the far away distances, the eye is only capable of detecting sooth variations (low frequency). In this project we are trying to use that phenomena to create an illusion by making hybrid image, which gets its low frequency content from an image and its high frequency content from another one.

Implementation of the convolutional filter

This function is used to perform convolution operation over RGB and gray scale image . It has two types of padding (zeros and mirror). It takes only odd shaped kernel, and returns the output in the form of an image with the same size as the input image

```

1 def my_imfilter(image, kernel, mode = 'zeros'):
2     # get the kernel dimesnsions in kh and kw
3     kh, kw = kernel.shape
4     # get the hight and width only as the third dimension may exist or
5     # not
6     ih = image.shape[0]
7     iw = image.shape[1]
8     # get the number of dimesnsions for both the kernel and the image
9     kdim = kernel.ndim
10    idim = image.ndim
11
12    assert kdim == 2, "kernel dimensions must be exactly two"
13    assert idim == 2 or idim == 3, "image dimensions must be exactly
14    two or three"
15    assert (kh%2) != 0 and (kw%2) != 0, "all kernel dimensions must be
16    odd"
17    # this foarmula calculates how many rows do i need to put so that i
18    # can make proper padding
19    hpad = (kh-1)//2
20    wpad = (kw-1)//2
21
22    filtered_image = np.zeros_like(image)
23
24    if mode == 'zeros':
25        md = 'constant'
26    elif mode == 'reflect':
27        md = 'reflect'
28    else:
29        raise Exception('the mode {} is not defined \n "zeros" and "
30        reflect are available"'.format(x))
31    # change the padding function according to the input image
32    if idim == 2:
33        padded_img = np.pad(image, [(hpad, hpad), (wpad, wpad)], mode=md)

```

```
29     else :
30         paddedImg=np.pad(image,[( hpad , hpad ) ,( wpad , wpad ) ,(0 ,0) ] ,mode='
31         constant')
32         dim_No=image.shape[2]
33         # apply convolution over the number of channels
34         for dim in range(0,dim_No):
35             for i in range(0,ih):
36                 for j in range(0,iw):
37                     # multiply the kernel with the cropped part of the image and
38                     # then sum the result
39                     cropped=paddedImg[ i : i+kh , j : j+kw , dim]
40                     filtered_image[ i , j , dim]=np.sum(np.multiply( kernel , cropped))
41     return filtered_image
```

Generation of Hybrid Image

In this part we are trying to add a low-pass filtered image and then add it to a high-pass filtered image of the same shape. This was implemented using 2 ways of filtering the first used the spatial convolution of the filter and the latter was done in frequency domain. The good thing of the frequency domain thing is that the convolution becomes a normal multiplication.

The kernel used is a Gaussian kernel with size 15x15 and $\sigma = 10$. This is how the low-pass filtered image was obtained and the high-filtered images was obtained by subtracting the low frequencies from the original image and this kept only high frequencies.

Here is a code snippet of the function implemented 2 times one with spatial convolution and the second using FFT-based convolution

```

1
2 def gen_hybrid_image(image1, image2, cutoff_frequency):
3
4     assert image1.shape == image2.shape
5
6     ksize = 15
7     sigma = cutoff_frequency
8
9
10    # Here I do outer product of 2 gaussian vectors to get 2D kernel
11    x = np.arange(-ksize//2, ksize//2+1)
12    gx = np.exp(-(x)**2/(2*sigma**2))
13    g = np.outer(gx, gx)
14    g /= np.sum(g)
15    kernel = g
16
17
18
19    # Your code here:
20    # looping over the channels of the image to apply the gaussian
    kernel
21    low_frequencies = np.zeros(image1.shape, dtype=np.float32)
22
23    for i in range(image1.shape[2]):
24        low_frequencies[:, :, i] = correlate2d(image1[:, :, i], kernel, 'same')
25        # Replace with your implementation
26
27    # (2) Remove the low frequencies from image2. The easiest way to do
    this is to
28    # subtract a blurred version of image2 from the original version
    of image2.
29    # This will give you an image centered at zero with negative
    values.
30
31    low_frequencies2 = np.zeros(image2.shape, dtype=np.float32)
32
33    for i in range(image1.shape[2]):
34        low_frequencies2[:, :, i] = correlate2d(image2[:, :, i], kernel, 'same')
35

```

```
36
37
38
39 high_frequencies = image2 - low_frequencies2 # Replace with your
    implementation
40
41
42 # print(np.sum(high_frequencies<0))
43
44 # (3) Combine the high frequencies and low frequencies
45
46 hybrid_image = low_frequencies/2 + high_frequencies/2 # Replace with
    your implementation
47
48 high_frequencies = np.clip(high_frequencies, -1.0, 1.0)
49 hybrid_image = np.clip(hybrid_image, 0, 1)
50
51 # np.clip(low_frequencies, 0, 1)
52 # np.clip(high_frequencies, 0, 1)
53 # np.clip(hybrid_image, 0, 1)
54 return low_frequencies, high_frequencies, hybrid_image
```

```
1 def gen_hybrid_image_fft(image1, image2, cutoff_frequency):
2     assert image1.shape == image2.shape
3
4     ksize = 15
5     sigma = cutoff_frequency
6
7     x = np.arange(-ksize//2, ksize//2+1)
8     gx = np.exp(-(x)**2/(2*sigma**2))
9     g = np.outer(gx, gx)
10    g /= np.sum(g)
11    kernel = g
12
13
14    # Applying the fft convolution on each channel
15    low_freqs = np.zeros(image1.shape)
16    for i in range(image1.shape[2]):
17        low_freqs[:, :, i] = fftconvolve(image1[:, :, i], kernel)
18
19
20
21    low_freqs2 = np.zeros(image2.shape)
22    for i in range(image1.shape[2]):
23        low_freqs2[:, :, i] = fftconvolve(image2[:, :, i], kernel)
24
25
26    # getting only high freqs of image2
27    high_freqs = image2 - low_freqs2
28
29
30    # combining the low freqs and high freqs
31    hybrid_image = low_freqs/2 + high_freqs/2 # Replace with your
32        implementation
33
34    high_freqs = np.clip(high_freqs, -1.0, 0.5)
35    hybrid_image = np.clip(hybrid_image, 0, 1)
36
37    return low_freqs, high_freqs, hybrid_image
```