Assignment 1. Image Filtering and Hybrid Image

1 Tổng quan bài tập lớn

Ở bài tập lớn này, các bạn sẽ hiện thực hàm tính tích chập (convolution) và sử dụng nó để sinh ra ảnh lai (hybrid). Kỹ thật sinh ảnh lai này được phát minh bởi nhóm nhà khoa học Oliva, Torralba, và Schyns năm 2006, và được đăng trên hội nghị SIGGRAPH (một trong những hội nghị rất uy tín trong lĩnh vực xử lý ảnh. Bằng cách pha trộn giữa tần số cao và thấp của 2 ảnh khác nhau, chúng ta có thể tạo ra ảnh lai giữa 2 ảnh ban đầu.

Yêu cầu hiện thực:

- Thực hiện lọc ảnh dùng **convolution** bằng hàm tự tạo
- Thực hiện lai ảnh **hybrid** với 2 ảnh có cùng kích thước
- Thực hiện lọc ảnh dựa vào phép biến đổi Fourier nhanh (**Fast Fourier Transform**)

2 Chi tiết hiện thực

2.1 Bộ lọc ảnh

Bộ lọc ảnh là một trong những công cụ phổ biến nhất trong việc xử lí ảnh. Việc kết hợp với những kernel khác nhau sẽ mang lại những kết quả có hiệu ứng khác nhau.

Bộ lọc ảnh có thể sử dụng phép tính tương quan (**correlation**) và tính tích chập (**convolution**). Trong bài tập lớn này ta sẽ thực hiện bộ lọc ảnh sử dụng phép tính tích chập.

2.1.1 Cơ sở lý thuyết

Lọc ảnh là một kỹ thuật dùng trong xử lí ảnh dùng để thay đổi một số tính chất của ảnh. Ví dụ: Bạn có thể làm mịn, làm sắc nét, nhấn mạnh những chi tiết của ảnh cũng như bỏ đi những chi tiết không mong muốn.

Hàm tính tích chập(convolution) là một giải thuật thường được dùng để thay đổi giá trị của một pixel dựa vào những giá trị của các pixel xung quanh nó. Dùng hàm tính tích chập để lọc ảnh, công thức là:

$$g[x, y] = \sum_{k,l} f[x - k, y - l]h[k, l] \tag{1}$$

f: ma trận đầu vào 2 chiều

h: ma trận kernel

g: ma trận đầu ra, có cùng kích thước với ma trận đầu vào

2.1.2 Ý tưởng hiện thực

Ta nhận thấy nếu thực hiện việc tính tích chập này với từng phần tử trong mảng thì ta sẽ mất khoảng 4 lần lặp để tính, điều đó sẽ mất rất nhiều thời gian. Do đó nếu lợi dụng những tính chất của ma trận như tích vô hướng thì sẽ tiết kiệm nhiều thời gian hơn.

Các bước làm chính:

- Ta phải mở rộng ảnh đầu vào, do khi dùng tích chập thì kernel sẽ cần dùng những giá trị bên ngoài biên. Theo đề thì ta sẽ mở rộng bằng giá trị 0

```
paddedImage = np.zeros((image.shape[0] + kernel.shape[0] - 1,
image.shape[1] + kernel.shape[1] -1))
paddedImage[int(kernel.shape[0] / 2) :
int(image.shape[0] + kernel.shape[0] / 2),
int(kernel.shape[1] / 2) :
int(image.shape[1] + kernel.shape[1] / 2)] = image
```

  Ta có thể dùng hàm **numpy.pad** để hiện thực việc padding.

- Tạo kernel đảo. Lí do cho việc này là ta có thể tận dụng được phép nhân ma trận để xử lí dữ liệu theo khối. Nên phần nào đó có thể nói hàm tính tích chập này được thực hiện dựa trên **correlation**

```
for i in range(len(kernel)):
    flipKernel[i] = kernel[i][::-1]

flipKernel = flipKernel[::-1]
```

  Ta cũng có thể làm việc này nhờ hàm **numpy.flip**

- Cuối cùng ta làm theo công thức với một ít thay đổi ở phần nhân ma trận

```
for i in range(image.shape[0]):
    for j in range(image.shape[1]):
        filteredImage[i, j] = (flipKernel *
        paddedImage[i:i+kernel.shape[0],
        j:j+kernel.shape[1]]).sum()
```

2.1.3 Kết quả

Sử dụng ảnh *bird.bmp*

- Sử dụng bộ lọc identity, bộ lọc này sẽ giữ nguyên ảnh ban đầu:

$$identity\_kernel = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Hình 1: *Trái:* Ảnh gốc. *Phải:* Ảnh sau khi lọc.

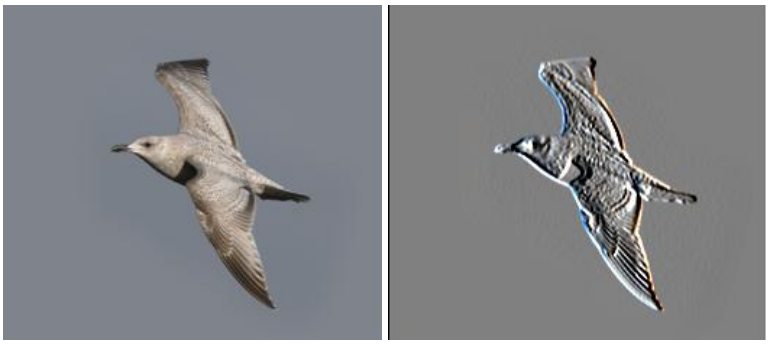
- Làm mờ nhẹ ảnh cùng với bộ lọc box filter:

$$box\_kernel = \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Hình 2: *Trái:* Ảnh gốc. *Phải:* Ảnh blur.

- Lọc ảnh dùng bộ lọc Sobel:

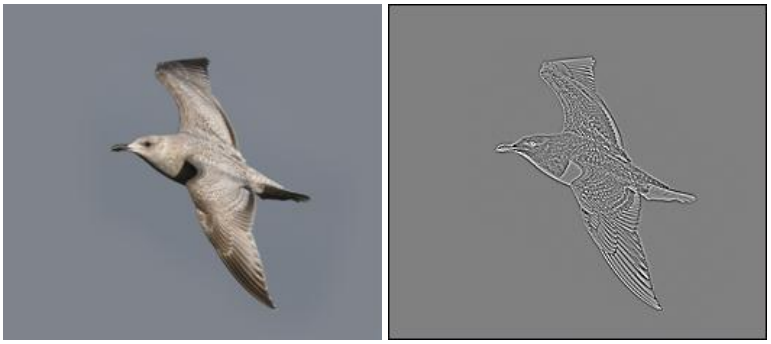
$$sobel\_kernel = \begin{bmatrix} -1 & 1 & 1 \\ -2 & 1 & 2 \\ -1 & 1 & 1 \end{bmatrix}$$



Hình 3: *Trái:* Ảnh gốc. *Phải:* Ảnh sau khi lọc sobel.

- Lọc ảnh dùng bộ lọc tần số cao(low-pass filter):

$$laplacian\_kernel = \begin{bmatrix} -1 & 1 & 1 \\ -2 & 1 & 2 \\ -1 & 1 & 1 \end{bmatrix}$$



Hình 4: *Trái:* Ảnh gốc. *Phải:* Ảnh sau khi lọc tần số cao.

Kết quả của những bộ lọc khác được tìm thấy trong thư mục *results*

2.2 Tạo ảnh lai hybrid

2.2.1 Cơ sở lý thuyết

Mỗi ảnh đều có những đặc điểm riêng, do đó nếu ta có thể gộp 2 ảnh làm một thì hiệu ứng đem lại sẽ mang đến những điều thú vị. Nhưng hai ảnh cùng tồn tại trong một ảnh như thế nào?

Nếu ta chuyển góc nhìn từ miền không gian sang miền tần số thì mỗi ảnh bao gồm rất nhiều tín hiệu cùng nhau tạo thành, và do mỗi tín hiệu đều được cấu thành từ một số lượng các tín hiệu sinusoidal nên ta có thể xác định được tần số của các tín hiệu đó.

Vậy ta có thể nói rằng, một ảnh bao gồm tập hợp của những tín hiệu có tần số khác nhau, do đó nếu gộp hai ảnh lại với nhau đơn giản là thay thế một số tín hiệu của ảnh A bằng một số tín hiệu khác của ảnh B. Mốc được lấy để chia cắt ảnh là ***cutoff_frequency***

2.2.2 Ý tưởng thực hiện

Dùng bộ lọc ảnh để lọc ảnh cho tần số thấp đi qua, ảnh còn lại cho tần số cao đi qua, sau đó cộng hai ảnh lại ta được một ảnh lai.

Các bước thực hiện

- Tạo kernel. Kernel trong trường hợp này là một **Gaussian Filter**, được dùng để cho tín hiệu tần số thấp đi qua.

```
s, k = cutoff_frequency, cutoff_frequency*2
probs = np.asarray([exp(-z*z/(2*s*s))/sqrt(2*pi*s*s)
for z in range(-k,k+1)], dtype=np.float32)
kernel2 = np.array(probs)
kernel1 = np.array([[item] for item in probs])
```

  Trong trường hợp này *cutoff_frequency* đóng vai trò là độ lệch chuẩn của hàm Gaussian. Để cho dễ hình dung thì sau khi chuyển qua miền tần số bằng biến đổi Fourier, hàm Gaussian sẽ có dạng một tương tự một hình tròn có bán kinh bằng với *cutoff_frequency*

  Ở đây theo đúng trình tự thì ta sẽ phải dùng tới một ma trận được tạo thành bằng cách nhân 2 ma trận Gaussian 1D

```
kernel = np.outer(probs, probs)
```

  Tuy nhiên, do ma trận này có tính đối xứng và 2 ma trận cấu thành đã được xác định nên ta có thể dùng tính chất của tính tích chập để giảm lượng tính toán

$$A = B * (C * D) \iff A = (B * C) * D \tag{2}$$

  Trong đó C,D là ma trận cột, hàng(1D)

  Do đó từ giờ thay vì dùng Gaussian filter để lọc ảnh thì ta dùng 2 lần lọc khác nhau.

- Dùng Gaussian(low pass) để lọc những tín hiệu của ảnh 1.

```
low_frequencies_inter = my_imfilter(image1, kernel1)
low_frequencies = my_imfilter(low_frequencies_inter, kernel2)
```

  Ở đây nếu dùng biến đổi Fourier ta có thể thấy ảnh sau khi lọc bao gồm tất cả những tín hiệu có tần số nằm gọn bên trong hình tròn bán kính *cutoff_frequency*

- Tạo ảnh high pass. Để tạo ảnh high-pass, tức cho tần số cao đi qua thì đơn giản nhất ta có thể dùng lại Gaussian filter để lọc những tín hiệu tần số thấp rồi sau đó lấy ảnh gốc trừ bớt đi ảnh vừa lọc.

```
    low_frequencies_inter_im2 = my_imfilter(image2, kernel1)
    low_frequencies_im2 = my_imfilter(low_frequencies_inter_im2,
                                      kernel2)
    high_frequencies = image2 - low_frequencies_im2
```

- Cuối cùng kết hợp 2 ảnh đã lọc để tạo thành ảnh lai hybrid

```
    hybrid_image = high_frequencies + low_frequencies
```

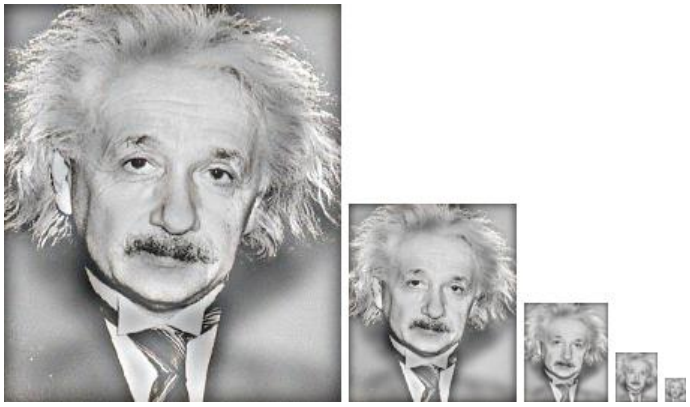
2.3 Kết quả

Việc lai ảnh được thực hiện trên 5 cặp ảnh trong *data*, với *cutoff_frequency* được dùng chung là 7.

- Với cặp ảnh chó mèo, ảnh chó sẽ được chọn để lọc bỏ tần số cao.

Hình 5: Ảnh chó và mèo sau khi lai

- Với cặp Marilyn và Einstein, đây là một cặp ảnh đã rất phổ biến trên internet trong việc gây ảo giác, và giờ ta có thể hiểu đó hoàn toàn là do xử lí ảnh.



Hình 6: Ảnh Marilyn và Einstein

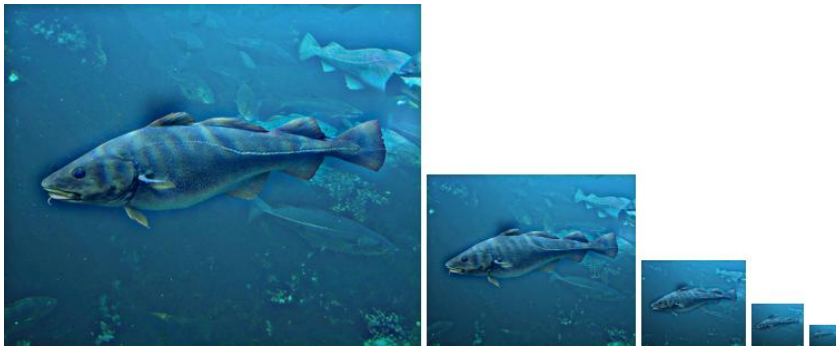
- Lai ảnh với cặp ảnh chim và máy bay

Hình 7: Ảnh chim máy bay

- Xe đạp và xe motor bản chất đã khá giống nhau

Hình 8: Ảnh xe đạp motor

- Cá và tàu ngầm là một lựa chọn thú vị



Hình 9: Ảnh cá tàu ngầm

3 Fast Fourier Transform

3.1 Lọc ảnh trên miền tần số

Cách tiếp cận dùng hàm tính tích chập là một cách tiếp cận thuộc về miền không gian, khi ta tương tác trực tiếp với những giá trị pixel và biến đổi chúng. Vậy liệu có cách tiếp cận nào gián tiếp hay không? Và nó sẽ có ích lợi gì hơn khi so với tính tích chập?

Miền tần số, như tên gọi của nó, sẽ là nơi tập hợp tất cả những tần số của các tín hiệu cấu thành nên ảnh. Ta có thể chuyển từ miền không gian sang miền tần số sử dụng biến đổi Fourier. Và ở miền tần số chúng ta có một tính chất quan trọng: phép tích chập được chuyển thành phép nhân. Khi đó phép tích chập sẽ được giải quyết gọn gàng hơn nhiều trong miền tần số.

Ý tưởng:

- Chuyển ảnh và kernel sang miền tần số
- Thực hiện phéo nhân giữa ảnh và kernel trên miền tần số
- Biến đổi ngược kết quả phép nhân về miền không gian

3.2 FFT - Biến đổi Fourier nhanh

3.2.1 Biến đổi Fourier

Biến đổi Fourier là một phép biến đổi giúp bạn chuyển một tín hiệu vào miền tần số. Biến đổi này được ứng dụng cực kì rộng rãi trong việc xử lý tín hiệu , đơn cử có thể kể đến lọc nhiễu, chỉnh sửa âm thanh,... Để hiểu rõ hơn về biến đổi Fourier và ý nghĩa của biểu thức biến đổi Fourier, bạn có thể tìm hiểu thêm tại đây.

Công thức cho biến đổi Fourier:

$$F(\omega) = \int_{-\infty}^{\infty} f(t) e^{-i\omega t} dt \tag{3}$$

Trên thực tế, người ta sử dụng biến đổi Fourier rời rạc(**DFT**):

$$x[k] = \sum_{n=0}^{N-1} x[n] e^{\frac{-j2\pi kn}{N}} \tag{4}$$

Biến đổi Fourier rời rạc trong không gian 2 chiều:

$$F(u,v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) e^{-j2\pi(\frac{ux}{M} + \frac{vy}{N})} \tag{5}$$

<Biểu thức đầy đủ có các hệ số $\frac{1}{M}, \frac{1}{N}$>

3.2.2 FFT

Biến đổi FFT thực chất chỉ là cách hiện thực nhanh của DFT, dựa vào việc nhận biết một số tính chất về tính tuần hoàn của biến đổi Fourier.

Ý tưởng FFT được sinh ra từ thuật toán **Divede and Conquer**, hay chia để trị. Bằng cách tách n thành 2 dãy con chẵn và lẻ, ta nhận thấy:

$$\begin{aligned} x[k] &= \sum_{n=0}^{N-1} x[n] e^{\frac{-j2\pi kn}{N}} \\ &= \sum_{r=0}^{\frac{N}{2}-1} x[2r] e^{\frac{-j2\pi k(2r)}{N}} + \sum_{r=0}^{\frac{N}{2}-1} x[2r] e^{\frac{-j2\pi k(2r+1)}{N}} \\ &= \sum_{r=0}^{\frac{N}{2}-1} x[2r] e^{\frac{-j2\pi k(r)}{N/2}} + e^{\frac{-j2\pi k}{N}} \sum_{r=0}^{\frac{N}{2}-1} x[2r] e^{\frac{-j2\pi k(r)}{N/2}} \\ &= x_{even}[k] + e^{\frac{-j2\pi k}{N}} x_{odd}[k] \end{aligned} \tag{6}$$

Ở đây ta thấy tính tuần hoàn, mọi giá trị n lớn hơn $\frac{N}{2}$ đều được lặp lại. Do đó có thể nói khối lượng tính toán đã được giảm đi một nửa. Và ta cứ tiếp tục cho đến khi nào không thể chia N được nữa thì ta dùng lại DFT.

3.2.3 Chi tiết hiện thực

- Khung sườn của FFT:

```
fourierImage = transformFFT2(padimage)
fourierKernel = transformFFT2(padkernel)

result = transformIFFT2(fourierImage * fourierKernel).real[:,
                                        :] / (padrows * padcols)
```

- Hiện thực transformFFT2, hàm FFT 2 chiều:
  Nếu nhìn theo góc nhìn khác từ phương trình:

$$\begin{aligned} F(u,v) &= \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) e^{-j2\pi(\frac{ux}{M} + \frac{vy}{N})} \\ &= \sum_{x=0}^{M-1} e^{-j2\pi \frac{ux}{M}} \sum_{y=0}^{N-1} f(x,y) e^{-j2\pi \frac{vy}{N}} \end{aligned} \tag{7}$$

  Từ đây ta nhận thấy FFT 2 chiều chỉ đơn giản là áp dụng FFT một chiều theo hàng rồi tiếp tục là FFT theo cột.

```
for u in range(nrows):
    firstDegreeImage[u] = transformFFT(inMatrix[u,:])

for v in range(ncols):
    fourierImage[:,v] = transformFFT(firstDegreeImage[:,v])
```

- Hiện thực transformFFT, hàm FFT 1 chiều:
  Hàm FFT có thể được hiện thực nhờ vào đệ quy, tuy nhiên sau nhiều lần chạy thì nhận thấy hàm này được viết chưa tối ưu bởi vì số lần lặp dãy lúy thừa của e là nhiều, do đó chạy rất chậm. Nên quyết định cuối cùng sẽ là dùng hàm có sẵn **numpy.fft.fft**.

```
def transformFFT(inArray):
return np.fft.fft(inArray)
length = len(inArray)

if length <= 64: return transformDFT(inArray)

evenArray = transformFFT(inArray[0::2])
oddArray = transformFFT(inArray[1::2])
expoCoeff = [exp( -2j * pi * k / length) * oddArray[k] for k
                                        in range(length // 2)]
return [evenArray[k]+expoCoeff[k] for k in range(length//2)]
                                        + \
    [evenArray[k] - expoCoeff[k] for k in range(length//2)]
```

- Hiện thực hàm chuyển đổi ngược FFT, hay transformIFFT2:
  Có nhiều cách để chuyển đổi ngược lại từ miền tần số, nếu muốn bạn có thể tham khảo tại đây

```
for i in range(nrows):
    copyMatrix[i, 1:] = copyMatrix[i, 1:][::-1]

for u in range(nrows):
    firstDegreeImage[u] = transformFFT(copyMatrix[u,:])

for i in range(ncols):
    firstDegreeImage[1:, i] = firstDegreeImage[1:, i][::-1]

for v in range(ncols):
    fourierImage[:,v] = transformFFT(firstDegreeImage[:,v])
```

3.2.4 So sánh hiệu quả

Dưới đây là kết quả chạy 2 hàm **my_imfilter** và **my_imfilter_fft** trên tập dữ liệu là ảnh *cat.bmp* và kernel là Gaussian filter với $\sigma = 7$.

```
image1 = load_image('../data/cat.bmp')
kernel = np.outer(probs, probs)
start = timeit.default_timer()
A = my_imfilter(image1, kernel)
print(timeit.default_timer() - start)
B = my_imfilter_fft(image1, kernel)
print(timeit.default_timer() - start)
print(np.allclose(A, B))
```

Các kết quả đều cho về True, tức là 2 hàm này đều cho ra kết quả giống nhau. Về thời gian chạy, tuy *timeit* không cho ra kết quả chính xác tuyệt đối nhưng có thể so sánh được về thời gian thực thi của 2 hàm.

my_imfilter	my_imfilter_fft
10.112	13.119
9.502	13.652
7.521	11.129
7.870	10.742
7.8	10.593
7.197	9.815

Ta có thể nhận thấy **my_imfilter** chạy nhanh hơn so với **my_imfilter_fft**. Điều này cũng dễ hiểu do dù cho theo lý thuyết việc dùng FFT sẽ cho kết quả nhanh hơn nhưng về mặt hiện thực dùng convolution sẽ gọn hơn và tránh nhiều vòng lặp. Mặt khác do việc xử lí ma trận khá nhiều trong FFT khiến cho hàm xử lí không được tối ưu,đó là vấn đề code còn nhiều sai xót cũng như hướng tiếp cận trong từng bước còn nhiều vấn đề. Nhìn chung cả 2 hướng tiếp cận đều chạy ổn và có thể chỉnh sửa trong tương lai.