UNIVERSIDAD DE SANTIAGO DE CHILE
FACULTAD DE INGENIERÍA
DEPARTAMENTO DE INGENIERÍA INFORMÁTICA

Convolución 2D

Integrantes: Gabriel Bustamante Toledo

Curso: Redes de computadores

Profesor: Carlos González

Ayudante: Nicole Reyes

5 de Agosto de 2020

Tabla de contenidos

Índice de figuras

Listings

1. Introducción

En el conjunto de las señales, también están incluidas las imágenes, que son en este caso señales en 2 dimensiones, las cuales están divididas en elementos mas pequeño que son denominados píxeles, los cuales representan la tonalidad de la imagen en esa posición tomando un valor entre 0 y 255. En imágenes con color, cada píxel posee tres valores que representan el rojo, verde y azul respectivamente (RGB). Para los efectos de esta experiencia se utilizará una imagen en escala de grises, por lo cual cada píxel solo poseerá un valor entre 0 y 255.

Para la presente actividad se espera que pueda ser aplicada la convolución en 2 dimensiones para cierta imagen, la cual es discreta y con valores finitos, esta imagen debe ser filtrada con distintos kernel para ver los efectos que le hace a dicha imagen.

1.1. Objetivos

1. Reforzar de forma práctica el procesamiento de señales.
2. Conocer como es la convolución en 2d.
3. Implementar de forma manual la convolución en 2d.
4. Detectar que tipo de filtro son ciertos kernel.

2. Marco teórico

Para comprender de que forma se llegará a la solución, hay que entender una parte teórica, para posteriormente poder utilizar herramientas informáticas. En este caso se pondrá en contexto que es la transformada de Fourier, que es la convolución en especifico en 2 dimensiones y como es que se aplica.

2.1. Convolución

Es un operador matemático que recibe dos funciones de entrada, mediante una integral si son continuas o una sumatoria si son discretas, da como resultado una tercera función. Para el caso de utilizarlo con las señales resulta ser un operador para poder aplicar filtros, en este caso será para aplicar filtros a una imagen. La formula a aplicar es la siguiente:

$$g[k,l] = f[k,l] * h[k,l] = \sum_{i=0}^{n} \sum_{j=0}^{m} f[i,j]h[i-k, j-l] \forall k, l \tag{1}$$

Lo cual a una forma practica y visual se traduce a la siguiente operación a realizar:

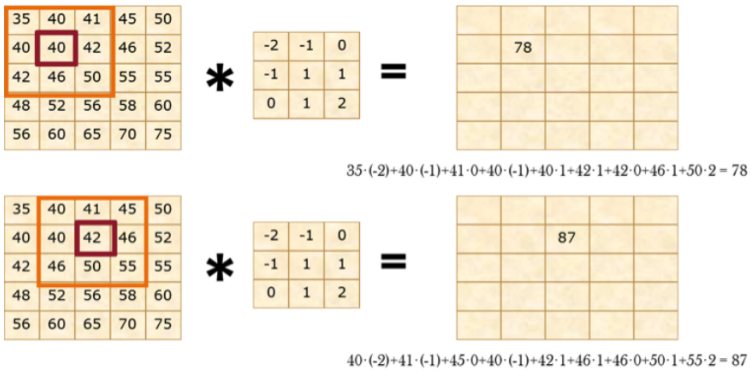

Figura 1: Visualización de la convolución en matrices

2.2. Transformada de Fourier

Es una transformación matemática utilizada para transformar señales desde el dominio en el que se encuentre normalmente hacia el dominio de la frecuencia o viceversa (en el caso de usarla de forma inversa), posee más aplicaciones en física e ingeniería, pero para nosotros es más relevante la propiedad de transformar las señales. La transformada de fourier de una matriz da como resultado otra matriz que indica cuales son las frecuencias presentes en ella, de lo cual se debe saber que los valores más cercanos al centro de la matriz son las frecuencias más bajas, y los valores más cercanos a los extremos de la matriz son las frecuencias más altas.

3. Desarrollo

3.1. Lectura de la imagen

Para la lectura de la imagen se utiliza la librería cv2 que proporciona la función imread() (código [1]), la cual recibe el nombre de la imagen que se abrirá y un argumento bandera para saber en que escala de colores se leerá la imagen, en este caso se utiliza un cero indicando que se leerá en escala de grises. La función devuelve la imagen en forma de matriz para poder utilizarla.

```
# Function that reads an image.
# Input: Image name, with its extension included.
# Output: Image in the form of a pixel matrix with values from 0 to 255.
def readImg(name):
    # Image name, 0 -> To read the grayscale image.
    img = cv2.imread(name, 0)
    return img
```

Listing 1: Lectura de imagen

3.2. Implementación de la convolución

Para esta experiencia es necesario implementar de forma manual la convolución entre matrices (código [2]). Para ello se crean dos funciones, la primera es aquella que se encarga de realizar las iteraciones e ir moviendo el kernel por cada píxel que debe modificar, dando como resultado una matriz del mismo tamaño que la original. La segunda función se encargar de operar cada elemento del kernel con los elementos acotados de la matriz, entregando como resultado un solo numero que es el píxel que debe ser modificado. Un punto a destacar es el problema que surge con los bordes de la nueva matriz, ya que en teoría es un poco mas pequeña, para resolver esto la matriz nueva queda con los bordes iguales a los de la matriz original.

```
# Function that performs the convolution between the kernel and the image to apply the filter.
# Input: Image and Kernel.
# Output: Filtered image.
def convolution(img, kernel):
    edge = int(len(kernel) / 2)
    filtered = copy.deepcopy(img)
    for i in range(len(img) - edge):
        for j in range(len(img[i]) - edge):
            if (i >= edge):
                if (j >= edge):
                    filtered[i][j] = operation(img, kernel, i - edge, j - edge)
    return filtered


# Function that executes the convolution operation.
# Input: Matrix, kernel, top right index of the image portion.
# Output: Number between 0 and 255.
def operation(img, kernel, i, j):
    acum = 0
    for k in range(len(kernel)):
        for l in range(len(kernel[k])):
            acum += (kernel[k][l] * img[i + k][j + l])
    if (acum < 0):
        return 0
    elif (acum > 255):
        return 255
    else:
        return acum
```

Listing 2: Implementación de la convolución

3.3. Pruebas

Al realizar de forma manual la aplicación de la convolución de matrices es muy probable que no quede perfecta desde un comienzo y se tienen que realizar pruebas para detectar las fallas que puede llegar a tener. Para esto se aplicó una pequeña prueba (código [3]), gracias a ello se logró detectar que se estaba copiando de forma incorrecta la matriz original a la matriz resultante, lo cual generaba datos falsos y muy elevados, lo cual fue corregido gracias a los test.

```
##### Test
    test = [[255, 2, 3, 4, 5, 6],
            [7, 8, 9, 10, 11, 12],
            [13, 14, 15, 16, 17, 18],
            [19, 20, 21, 22, 23, 24],
            [25, 26, 27, 28, 29, 30]]

    kernelTest = [[0, 1, 0],
                  [0, 1, 0],
                  [0, 1, 0]]

    filterTest = convolution(test, kernelTest)

    plt.imshow(filterTest, cmap='gray')
    plt.show()
```

Listing 3: Pruebas aplicadas

3.4. Aplicación de la convolución

En esta ocasión se utilizará un kernel para aplicar un suavizado gaussiano, y un kernel que servirá como filtro detector de bordes (código [4]), los cuales serán aplicados a la imagen original (figura [2]). Lo cual nos entregó dos imágenes resultantes, la primera es una imagen un poco borrosa, casi imperceptible la diferencia (figura [3]). La segunda imagen ya es bastante mas distinta, en la cual es fácil darse cuenta que solo los bordes están resaltados con blanco (figura [4]).

```
# Matrix for applying Gaussian blur.
    kernelBlur = [[1, 4, 6, 4, 1],
                  [4, 16, 24, 16, 4],
                  [6, 24, 36, 24, 6],
                  [4, 16, 24, 16, 4],
                  [1, 4, 6, 4, 1]]
    kernelBlur = normalization(kernelBlur)

    # Edge detection matrix.
    kernelEdge = [[1, 2, 0, -2, -1],
                  [1, 2, 0, -2, -1],
                  [1, 2, 0, -2, -1],
                  [1, 2, 0, -2, -1],
                  [1, 2, 0, -2, -1]]
```

Listing 4: Kernel a utilizar

Figura 2: Imagen original

Figura 3: Imagen con suavizado gaussiano

3.5. Transformada de Fourier

Para el calculo de la transformada de fourier se hace uso de la librería numpy, en especifico la función fft.fft2(), que es quien hace el calculo para las matrices. Dando como resultado una matriz de frecuencias. En esta experiencia se le calcula la transformada de fourier a la imagen original, la imagen con suavizado gaussiano y la imagen con detección de bordes, de esta forma analizar que tipo de filtro son cada uno de los kernel aplicados.

Figura 4: Imagen con detección de bordes

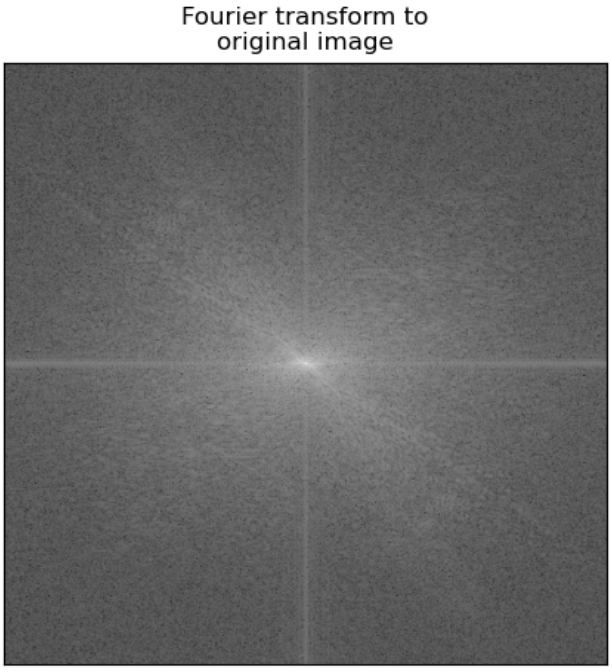

Figura 5: Transformada de fourier a la imagen original

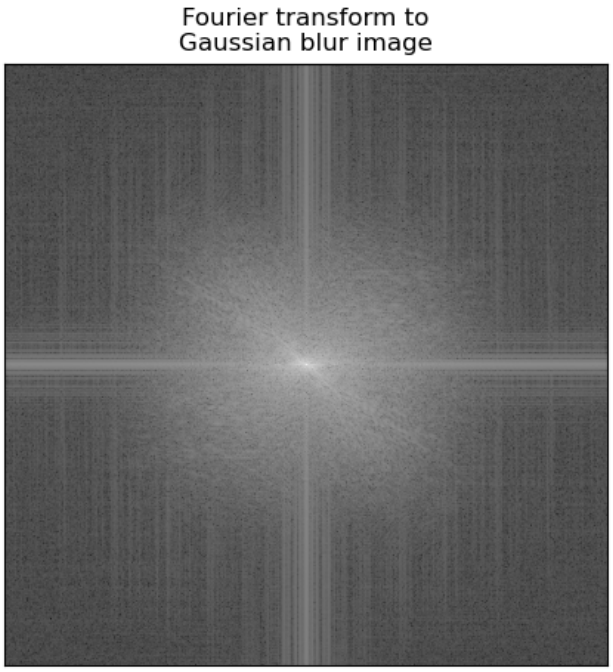

Figura 6: Transformada de fourier a la imagen con suavizado gaussiano

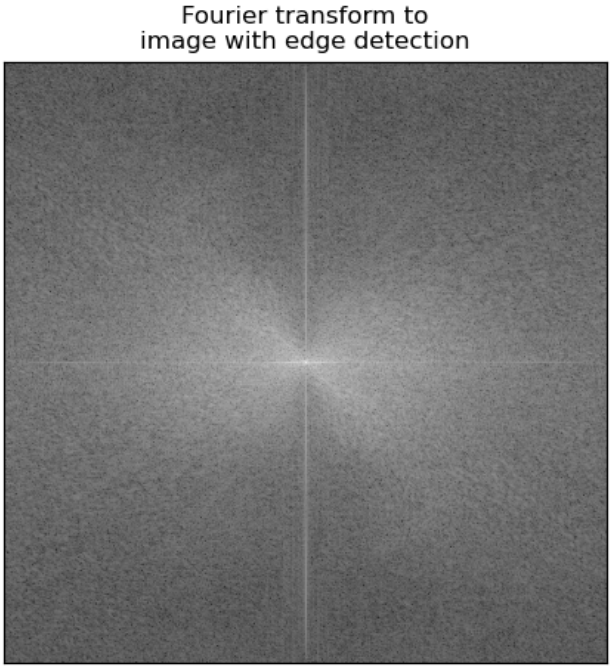

Figura 7: Transformada de fourier a la imagen con detección de bordes

4. Análisis

Las imágenes resultantes al aplicar la convolución con ambos kernel resulta ser una respuesta bastante esperable, ya que se logra tener una imagen un poco suavizada y adicionalmente de puede obtener una imagen con detección de bordes, uno de los problemas que se obtenía inicialmente era que algunos valores resultantes de la convolución eran menores a 0 o mayores a 255, esto fue solucionado con una pequeña restricción de los valores de salida de la función involucrada en este fallo. Visualmente estos resultados están en lo correcto ya que obtenemos lo que esperamos, pero no podemos ver mas allá o en mas detalle el resultado, entonces ¿están por completo correcto estos resultados? o ¿tienen sentido?, para responder esto es necesario saber dos cosas, lo primero es a que tipo de filtro corresponde cada kernel aplicado y cual es la transformada de fourier de la imágenes filtradas. Para lo primero se sabe que un kernel de suavizado gaussiano aplica como un filtro pasa bajos, por lo cual si se le aplica una transformada de fourier a la imagen con este suavizado se tendría que apreciar una disminución en las frecuencias más altas (más a las orillas), y que se mantengan en cierta forma la frecuencias mas bajas, esto es exactamente lo que se ha obtenido (figura [6]), por lo tanto tiene mucho sentido. Para el kernel de detección de bordes, también conocido como filtro de tipo sobel es un filtro pasa altos, por lo cual se debería ver una transformada de fourier con los bordes mucho mas claros mostrando una mayor presencia de frecuencia en aquellos lugares, si vemos la transformada obtenida (figura [7]) y la comparamos con la transformada de la imagen original (figura [5]), se logra aprecias que efectivamente hay una mayor presencia de frecuencias altas en la transformada de la detección de bordes, por ello se puede llegar a la conclusión de que los valores obtenidos para ambos kernel son los esperados y tiene mucho sentido.

5. Conclusión

A lo largo de esta experiencia se puede ver que uno de los principales objetivo, o en lo que estaba mas centrado el desarrollo de la actividad, es en torno a entender en como es que es aplicada la convolución y probar si verdaderamente lo hemos hecho bien, esta comprobación añade un análisis adicional, que es darse cuenta a que tipo de filtro corresponde cada kernel y comprobar si todo lo anterior ha estado en lo correcto con una transformada de fourier. Durante el desarrollo se encontraron problemas relacionados a la naturaleza de lenguaje, un poco relacionado con el manejo de la memoria en cuando a como es que copia las matrices lo cual pudo ser solucionado fácilmente leyendo documentación de las matrices en python, también problemas totalmente esperables en cuando a la función de convolución ya que puede llegar a ser un poco engorroso el calculo.

Personalmente rescato el hecho de poder aprender a utilizar distintos kernel para la manipulación de imágenes en este caso. En un comienzo creía que al aplicar esos filtros, un poco raros y tan pequeños comparados a la imagen, la imagen cambiaría por completo o llegaría a ser muy poco reconocible. Además a esto añado lo importante que ha sido interiorizarme cada vez más en este lenguaje python.

Bibliografía

[1] [Online] https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_gui/py_image_display/py_image_display.html.

[2] [Online] https://riunet.upv.es/bitstream/handle/10251/69639/4524-15767-1-PB.pdf.

[3] [Online] http://www2.elo.utfsm.cl/~elo328/pdf1dpp/PDI09_Frecuencia_1dpp.pdf.

[4] [Online] https://docs.opencv.org/master/d4/da8/group__imgcodecs.html#ga288b8b3da0892bd651fce07b3bbd3a56.

[5] [Online] https://www.lpi.tel.uva.es/~nacho/docencia/ing_ond_1/trabajos_03_04/sonificacion/cabroa_archivos/pasoalto.html.