CC3182 – Visión por Computadora

Laboratorio 1

Link del respositorio: https://github.com/faguilarleal/lab1_vision

Integrantes:

- Francis Aguilar
- César Lopez
- Jose Marchena

Usted ha sido contratado como Ingeniero de Visión Computacional Junior en "AutonoVision", una startup que desarrolla sistemas de navegación para robots de almacén. El equipo de hardware ha instalado nuevas cámaras, pero las imágenes llegan con mucho ruido térmico debido a las condiciones de luz del almacén. El módulo actual de detección de obstáculos está fallando: detecta la textura del suelo de concreto como si fueran "bordes" de obstáculos, frenando el robot innecesariamente. Su Director de Proyecto le ha asignado la tarea de construir un pipeline de pre-procesamiento robusto desde cero para entender el problema a nivel matemático y ajustar los parámetros óptimos para el despliegue.

Task 1 - Análisis Teórico y Analítico

Considerando el escenario previamente planteado, conteste:

1. Su jefe sugiere usar un filtro de media (Box Filter) de 7x7 para eliminar el ruido rápido. Usted cree que es una mala idea. Explique matemáticamente y con un diagrama visual (dibujado) por qué un Box Filter de ese tamaño es perjudicial para la detección precisa de la posición de un obstáculo comparado con un filtro Gaussiano del mismo tamaño.

R

Un Box Filter asi de grande aplica un alizamiento extremo sobre las imagenes que procesa y causa perdida en el contraste entre bordes y el fenomeno de Ringing en las imagenes luego de un par de iteraciones. Ademas de los artefactos que causa, el filtro de media tiende a perder detalle de formas en su alizamiento, teniendo un sesgo a hacer ver todo cuadrado, lo que impediria la deteccion precisa de obstaculos ya que complicaria su deteccion optima

Mientras tanto, un filtro gausiano si mantiene la curvatura de las imagenes en su alizamiento, lo que puede llevar a mejores resultados en la deteccion de objetos

Filtro Gausiano: 

Filtro de Media 

1. Al realizar la convolución en los bordes de la imagen (por ejemplo, en el píxel 0,0), el kernel "se sale" de la imagen.

A. Si el robot navega por pasillos oscuros con luces brillantes al final, ¿por qué el Zero- Padding podría generar falsos positivos de bordes en la periferia de la imagen?

- El Zero Padding introduce valores artificiales (0) fuera de la imagen. En escenarios como pasillos oscuros con luces brillantes al fondo, esto provoca un salto brusco de intensidad entre el borde real de la imagen y el padding. Esto hace que el robot pueda detectar obstaculos falsos cerca de los bordes de la imagen.

B. ¿Qué estrategia de padding (Reflect, Replicate, Wrap) recomendaría para evitar esto y por qué?

- Es mejor reflect, ya que mantiene la continuidad de intensidad, por ejemplo, se usa de referencia lo que ya estaba antes, asi no hay posibilidad de bordes o contraste inesperado.

1. Dada la siguiente sub-imagen I de 3x3 y el kernel K:

$$ I = \begin{pmatrix} 10 & 10 & 10 \\ 10 & 10 & 10 \\ 0 & 10 & 10 \end{pmatrix} $$

$$ K = \begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix} $$

a. Calcule el valor del píxel central resultante de la convolución

R r = 10*0* + *10*1 + 10*0* + *10*1 + 10-*4* + *10*1 + 10*0* + *10* + *10*0 = 10 + 10 - 40 + 10 + 10 = 0

El producto punto entre estas dos matriceses 0.

b. ¿Qué tipo de estructura detecta este filtro K (conocido como Laplaciano)?

R

Similar al filtro de sobel, el laplaciano detecta cambio drastico de un pixel con sus vecinos. La diferencia es que el Laplaciano detecta cambios o bordes sin importar la direccion, pues responde igual a todos.

Task 2 – Práctica

Ejercicio 1: Convolución 2D Genérica

Escriba una función mi_convolucion(imagen, kernel, padding_type='reflect'), Considerando lo siguiente:

- Restricción 1: La función debe manejar imágenes en escala de grises.
- Restricción 2: Debe implementar el padding manualmente antes de operar.
- Reto de optimización: Intente no usar 4 bucles for anidados. Investigue cómo usar slicing de NumPy o np.einsum para hacerlo vectorizado, o al menos reduzca a 2 bucles.
- Nota: Recuerde que matemáticamente la convolución invierte el kernel. Implemente el "flip" del kernel dentro de la función.

```
from PIL import Image
import numpy as np
from matplotlib import pyplot as plt

def mi_convolucion(img, kernel, padding_type='reflect'):
    kernel = np.flip(kernel)
```

```
    raw_array = np.array(img.convert('L'), dtype=np.float32)
    k_h, k_w = kernel.shape

    if padding_type == 'reflect':
        img_array = np.pad(raw_array, pad_width=k_h//2,
        mode='reflect')
    elif padding_type == 'zeros':
        img_array = np.pad(raw_array, pad_width=k_h//2)
    else:
        img_array = raw_array.copy()

    o_h = img_array.shape[0] - k_h + 1
    o_w = img_array.shape[1] - k_w + 1

    output = np.zeros((o_h, o_w), dtype=np.float32)

    for i in range(o_h):
        for j in range(o_w):
            window = img_array[i:i+k_h, j:j+k_w]
            output[i, j] = np.sum(window * kernel)

    return output

# Ejemplo
test = Image.open('./test.png')
# Box shaddow kernel for test
kernel = np.ones((30,30))/900

# show
rslt = mi_convolucion(test, kernel, padding_type='reflect')
plt.imshow(rslt, cmap='gray')
plt.show()
plt.imshow(test, cmap='gray')
plt.show()
```

Ejercicio 2: Generador de Gaussianos

Escriba una función generar_gaussiano(tamano, sigma). Para ello considere: • La función debe devolver una matriz cuadrada de tamano x tamano con los coeficientes de una distribución Gaussiana 2D centrada. • Importante: Asegúrese de que la suma de todos los elementos de la matriz sea igual a 1.0 (Normalización).

```
import numpy as np

def generar_gaussiano(tamano, sigma):
    assert tamano % 2 == 1, "El tamaño debe ser impar"

    k = tamano // 2
    x, y = np.meshgrid(np.arange(-k, k+1), np.arange(-k, k+1))

    gauss = np.exp(-(x**2 + y**2) / (2 * sigma**2))

    # Normalizacion
    gauss = gauss / np.sum(gauss)

    return gauss
```

Ejercicio 3: Pipeline de Detección de Bordes (Sobel)

Cree una función detectar_bordes_sobel(imagen).. Para ello considere: • Aplique los kernels de Sobel *Gx* y *Gy* usando su función mi_convolucion • Calcule y retorne dos matrices: o Magnitud del gradiente:

$$ G = \sqrt{ Gx^2 + Gy^2} $$

2 (Normalizada a 0-255 para visualizar) o Dirección del gradiente: θ=arctan2(Gy,Gx) (En radianes o grados)

```
def detectar_bordes_sobel(img):
    SOBEL_GX = np.array([
        [-1, 0, 1],
        [-2, 0, 2],
        [-1, 0, 1]
    ], dtype=np.float32)

    SOBEL_GY = np.array([
        [-1, -2, -1],
```

```
        [ 0,  0,  0],
        [ 1,  2,  1]
    ], dtype=np.float32)

    Gx = mi_convolucion(img, SOBEL_GX)
    Gy = mi_convolucion(img, SOBEL_GY)

    G = np.sqrt(Gx**2 + Gy**2)
    G_norm = (G / G.max()) * 255
    G_norm = G_norm.astype(np.uint8)

    theta = np.arctan2(Gy, Gx)

    return G_norm, theta

from PIL import Image

img = Image.open("imagen.jpg")

G, theta = detectar_bordes_sobel(img)

# Visualización
plt.figure(figsize=(12,4))

plt.subplot(1,3,1)
plt.title("Original")
plt.imshow(img, cmap='gray')
plt.axis('off')

plt.subplot(1,3,2)
plt.title("Magnitud del gradiente")
plt.imshow(G, cmap='gray')
plt.axis('off')

plt.subplot(1,3,3)
plt.title("Dirección del gradiente")
plt.imshow(theta, cmap='gray')
plt.axis('off')

plt.show()
```

Task 3 – Evaluación de Ingenería y Criterio

Experimento A: El efecto de Sigm (σ)

Cargue una imagen con ruido (agregue ruido "Sal y Pimienta" o Gaussiano artificialmente a una foto limpia si es necesario).

1. Genere 3 versiones de detección de bordes (Magnitud Sobel) variando el pre-procesamiento Gaussiano:

a. Sin suavizado.

b. Gaussiano $\sigma = 1$ (kernel sugerido 5x5).

c. Gaussiano $\sigma = 5$ (kernel sugerido 31x31).

```
img = Image.open("ruido.jpg")

G_a, _ = detectar_bordes_sobel(img)

img = Image.open("ruido.jpg")
gauss_5 = generar_gaussiano(5, 1)

img_suave_1 = mi_convolucion(img, gauss_5)

img_suave_1_pil = Image.fromarray(img_suave_1.astype(np.uint8))

G_b, _ = detectar_bordes_sobel(img_suave_1_pil)

img = Image.open("ruido.jpg")
gauss_31 = generar_gaussiano(31, 5)

img_suave_5 = mi_convolucion(img, gauss_31)
img_suave_5_pil = Image.fromarray(img_suave_5.astype(np.uint8))

G_c, _ = detectar_bordes_sobel(img_suave_5_pil)

plt.figure(figsize=(15,5))
```

```
plt.subplot(1,3,1)
plt.imshow(G_a, cmap='gray')
plt.title("Sin suavizado")
plt.axis('off')

plt.subplot(1,3,2)
plt.imshow(G_b, cmap='gray')
plt.title("Gaussiano σ = 1 (5x5)")
plt.axis('off')

plt.subplot(1,3,3)
plt.imshow(G_c, cmap='gray')
plt.title("Gaussiano σ = 5 (31x31)")
plt.axis('off')

plt.show()
```

1. Análisis: Muestre las tres imágenes de bordes resultantes. ¿Qué pasa con los bordes finos cuando σ es muy alto? ¿Qué pasa con la textura del suelo cuando no hay suavizado? Como ingeniero, ¿cuál elegiría para detectar pallets grandes ignorando grietas pequeñas en el suelo?

Cuando σ es muy alto los bordes se hacen mas gruesos y se pierde la nitidez de la imagen y se distorciona bastante, son bordes muy ruidosos y la respuesta a pequeños o irrelevantes cambios es muy alto. Cuando se suaviza la imagen antes de procesarla se reduce bastante las variaciones pequeñas del suelo, los bordes principales se conservan y se mantiene un buen compromiso entre detalle y limpieza. Los bordes finos se atenúan o desaparecen, porque el suavizado Gaussiano elimina las componentes de alta frecuencia asociadas a detalles pequeños. La textura genera falsos bordes, aumentando el ruido y dificultando la segmentación de objetos relevantes.

Eligiría el segundo suavizado con σ = 5 porque un σ alto elimina grietas y textura, conservando solo bordes estructurales relevantes

Experimento B: Histéresis Manual (Simulación de Canny)

Usted ha calculado la Magnitud del Gradiente en el paso 3.3. Ahora implemente una función simple de umbralización umbral_simple(magnitud, T) y compare visualmente con cv2.Canny.

```
def umbral_simple(magnitud, T):
    thresh_matrix = np.where(magnitud > T,1, 0)
    return np.astype(thresh_matrix, np.uint8)

test = Image.open('./test.png').convert('L')

Gx = np.array([[-1, 0, 1],
               [-2, 0, 2],
               [-1, 0, 1]])
Gy = np.array([[-1, -2, -1],
               [ 0,  0,  0],
               [ 1,  2,  1]])

Mx = mi_convolucion(test, Gx, padding_type='reflect')
My = mi_convolucion(test, Gx, padding_type='reflect')
M = np.multiply(Mx, Mx) + np.multiply(My, My)
M = np.sqrt(M)

import cv2
fig, ax = plt.subplots(1, 3, figsize=((10,5)))

ax[0].imshow(M)
ax[0].set_title("Magnitud")

T = umbral_simple(M, 50)
ax[1].imshow(T)
ax[1].set_title("Umbral Simple (50)")

img = np.array(test, dtype=np.uint8)
C = cv2.Canny(img, threshold1=50, threshold2=50)
ax[2].imshow(C)
ax[2].set_title("Cv2-Canny (50)")

Text(0.5, 1.0, 'Cv2-Canny (50)')
```

1. Intente encontrar un valor T único que limpie el ruido pero mantenga los bordes.
2. Observe el resultado: ¿Se rompen las líneas de los bordes?

- Sí. Al aplicar un umbral simple sobre la magnitud del gradiente se observa que los bordes se ven un poco fragmentados, con unas pequeñas variaciones en la intensidad causan como cortes en lineas continuas

1. Pregunta Crítica: Explique por qué un simple umbral de corte (Thresholding) nunca será tan efectivo como el método de Histéresis usado en Canny.

- El umbral simple decide de forma local e independiente para cada píxel, lo cual lo hace muy poco efectivo, mientras que el metodo e histeresis que se usa en canny emplea dos umbrales (alto y bajo), conserva los bordes debiles solo si estan conectados a bordes fuertes.

¿Qué problema específico resuelve la conectividad de la histéresis en el contexto de un robot moviéndose y vibrando (lo que causa cambios leves de iluminación en los bordes)?

- En un robot en movimiento hay vibraciones, cambios de iluminacion y de gradiente, la histeresis permite que el sistema no pierda bordes importantes por pequeñas variaciones y ayuda a que detecte obstaculos como estructuras completas, y no fragmentadas.