**IBM CLOUD DEVELOPMENT PROJECT**

**A blue and white striped logo

Description automatically generated**

**COLLEGE CODE: 4224**

**UNIVERSITY COLLEGE OF ENGINEERING TINDIVANAM**

**MELPAKKAM 604001.**

**IMAGE RECOGNITION WITH IBM CLOUD VISUAL RECOGNITION**

**PROJECT TITLE FOR IMAGE RECOGNITION IS TO**

**IMAGE EDGE DETECTION**

**SUBMITTED BY**

**G M PRANAV**

**B SHAHID AFRIDI**

**G DEVA KUMAR**

**S SUDHAGARAN**

**R MANIGANDAN**

**INTRODUCTION:**

Image edge detection is a fundamental technique in computer vision, and its application in cloud development has opened new horizons for a wide range of industries. By identifying and highlighting the edges of objects within an image, this process allows for enhanced feature extraction, object recognition, and image segmentation. In the context of cloud application development, integrating edge detection algorithms into cloud-based systems provides scalability, accessibility, and real-time processing capabilities. This introduction sets the stage for exploring how image edge detection can empower cloud applications by improving image analysis, security, and automation, among other use cases.

**LIBRARIES FOR IMAGE EDGE DETECTION:**

1. Open cv

2. Numpy

3. Matplotlib

**OPEN CV:**

Import OpenCV:

Begin by importing the OpenCV library in your Python code:

python

import cv2

Read an Image:

Use the `cv2.imread()` function to load the image you want to process. For example:

python

image = cv2.imread('image.jpg', cv2.IMREAD\_GRAYSCALE)

Ensure that you read the image in grayscale (`cv2.IMREAD\_GRAYSCALE`) to simplify edge detection.

Select an Edge Detection Algorithm:

OpenCV provides various edge detection algorithms. Common ones include Canny, Sobel, and Laplacian. Choose the most suitable algorithm for your application.

Apply Edge Detection :

Use the chosen algorithm to detect edges in the image. For example, using the Canny edge detector:

python

edges = cv2.Canny(image, threshold1, threshold2)

Adjust the `threshold1` and `threshold2` values to control the sensitivity of edge detection.

Display the Edge Image:

You can display the resulting edge-detected image using `cv2.imshow()`:

python

cv2.imshow('Edges', edges)

cv2.waitKey(0)

cv2.destroyAllWindows()

Save the Edge Image:

If you want to save the edge-detected image, use `cv2.imwrite()`:

python

cv2.imwrite('edges.jpg', edges)

Tune Parameters:

Experiment with different algorithm parameters to fine-tune edge detection for your specific image and requirements. Parameters like kernel size, aperture size, and thresholds can significantly impact the results.

Pre-processing:

Sometimes, it's beneficial to perform pre-processing on the image, such as blurring or noise reduction, before applying edge detection. OpenCV provides functions for these tasks, like `cv2.GaussianBlur()`.

Performance:

Keep in mind that the choice of algorithm and parameters can affect the computational performance of your edge detection. Consider the trade-off between accuracy and processing speed.

Further Processing:

Edge detection is often a preprocessing step for more advanced computer vision tasks, like object detection and image segmentation. Use the detected edges as a foundation for these tasks.

**NUMPY:**

NumPy for Array Manipulation:

NumPy is a fundamental library for numerical computing in Python. It provides a powerful array object, `numpy.ndarray`, which is essential for storing and manipulating image data.

Image Data Representation:

Images are typically represented as NumPy arrays, with pixel values stored as elements of these arrays. Grayscale images are 2D arrays, while color images are often 3D arrays with dimensions for height, width, and color channels (e.g., RGB).

Applying Filters:

NumPy allows you to apply convolution filters for edge detection. You can define custom kernels or use built-in functions like `numpy.convolve` to perform operations like the Sobel filter or the Prewitt filter.

Gradient Calculation:

Edge detection relies on calculating gradients to find areas of abrupt intensity change. You can use NumPy's gradient calculation functions like `numpy.gradient` to compute the gradient magnitude and direction in an image.

Thresholding:

Once you have gradient information, NumPy can be used to perform thresholding to highlight edges by setting certain pixel values to black or white based on their gradient magnitude.

Image Visualization:

While NumPy is primarily for numerical computation, you can use other libraries like Matplotlib to visualize images and the results of edge detection operations.

Performance Considerations:

NumPy operations can be memory-intensive. For large images, it's essential to be mindful of memory usage and consider data types (e.g., `uint8`, `float32`) to balance precision and performance.

Loop-Free Coding:

NumPy encourages a vectorized, loop-free coding style, which can improve the efficiency of edge detection algorithms when applied to entire images or image regions.

Integration with Other Libraries:

NumPy is often used in conjunction with other libraries like OpenCV or scikit-image, which provide high-level image processing functions along with NumPy's array manipulation capabilities.

NumPy and Python Ecosystem:

NumPy is an integral part of the broader scientific Python ecosystem, making it easier to integrate image edge detection with other data analysis and visualization tools.

**MATPLOTLIB:**

Matplotlib is a popular Python library for creating visualizations, including displaying and saving images.

Displaying Images:

Matplotlib can be used to display images before and after edge detection. You can load images using libraries like OpenCV or PIL and then use Matplotlib to visualize them using functions like `imshow()`.

Subplots:

You can create multiple subplots to display the original image, the result of edge detection, and other intermediate steps. This helps in comparing and analyzing the impact of different edge detection algorithms.

Color Maps:

Matplotlib provides various color maps (colormaps) that can be applied to the images. For example, you can use a grayscale colormap to display edge-detected images in a visually appealing way.

Customizing Plots:

You can customize plots with titles, labels, legends, and color bars to provide context and explanations for the images and their features.

Saving Images:

Matplotlib allows you to save the plots as image files, such as PNG or JPEG. This is useful for documenting and sharing the results of edge detection.

Interactive Exploration:

Matplotlib can be used in interactive environments like Jupyter Notebooks, which can aid in real-time exploration of different edge detection parameters and techniques.

For example:

import matplotlib.pyplot as plt

import cv2

image = cv2.imread('your\_image.jpg', cv2.IMREAD\_COLOR)

plt.subplot(121)

plt.imshow(cv2.cvtColor(image, cv2.COLOR\_BGR2RGB))

edges = cv2.Canny(image, 100, 200)

plt.subplot(122)

plt.imshow(edges, cmap='gray')

plt.show()

In this example, `imshow()` is used to display both the original image and the edge-detected version. Matplotlib's flexibility and integration with other image processing libraries make it a valuable tool for visualizing the results of image edge detection.

**PYTHON CODE:**

from PIL import Image

import numpy as np

import cv2

def Canny\_Deriche\_edge\_detection(img\_name):

img = Image.open(img\_name)

gray = img.convert('L')

edges = np.zeros(gray.size)

for i in range(1, gray.size[0]-1):

for j in range(1, gray.size[1]-1):

dx = int(gray.getpixel((i, j+1))) - int(gray.getpixel((i+1, j)))

dy = int(gray.getpixel((i, j+1))) - int(gray.getpixel((i, j)))

edges[i,j] = np.sqrt(dx\*dx + dy\*dy)

edges = np.rot90(edges, k=3)

new\_img = Image.fromarray(edges)

return new\_img

def frei\_chen(img\_name):

img = cv2.imread(img\_name)

gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

fc\_x = np.array([[-1, 0, 1], [-np.sqrt(2), 0, np.sqrt(2)], [-1, 0, 1]])

fc\_y = np.array([[-1, -np.sqrt(2), -1], [0, 0, 0], [1, np.sqrt(2), 1]])

edges\_x = cv2.filter2D(gray, cv2.CV\_64F, fc\_x)

edges\_y = cv2.filter2D(gray, cv2.CV\_64F, fc\_y)

edges = np.sqrt(np.square(edges\_x) + np.square(edges\_y))

return edges

def log\_edge\_detection(img\_name):

img = cv2.imread(img\_name)

gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

edges = cv2.Laplacian(gray, cv2.CV\_64F, ksize=5)

return edges

def prewitt\_edge\_detection(img\_name):

img = cv2.imread(img\_name)

gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

edges\_x = cv2.Sobel(gray, cv2.CV\_64F, 1, 0, ksize=3)

edges\_y = cv2.Sobel(gray, cv2.CV\_64F, 0, 1, ksize=3)

edges = np.sqrt(np.square(edges\_x) + np.square(edges\_y))

return edges

def canny\_edge\_detection(img\_name):

img = cv2.imread(img\_name)

gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

edges = cv2.Canny(gray, 100, 200)

return edges

def sobel\_edge\_detection(img\_name):

img = cv2.imread(img\_name)

gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

edges = cv2.Sobel(gray, cv2.CV\_64F, 1, 0, ksize=5)

return edges

class EdgeDetector:

def \_\_init\_\_(self, img\_name)

self.img = cv2.imread(img\_name)

self.gray = cv2.cvtColor(self.img, cv2.COLOR\_BGR2GRAY)

def detect\_edges\_prewitt(self,ksize=3):

edges\_x = cv2.Sobel(self.gray, cv2.CV\_64F, 1, 0, ksize=ksize)

edges\_y = cv2.Sobel(self.gray, cv2.CV\_64F, 0, 1, ksize=ksize)

edges = np.sqrt(np.square(edges\_x) + np.square(edges\_y))

return edges

def detect\_edges\_canny(self):

edges = cv2.Canny(self.gray, 100, 200)

return edges

def detect\_edges\_sobel(self,ksize=5):

edges = cv2.Sobel(self.gray, cv2.CV\_64F, 1, 0, ksize=ksize)

return edges

def detect\_edges\_log(self,ksize=5):

edges = cv2.Laplacian(self.gray, cv2.CV\_64F, ksize=ksize)

return edges

def detect\_edges\_frei\_chen(self):

fc\_x = np.array([[-1, 0, 1], [-np.sqrt(2), 0, np.sqrt(2)], [-1, 0, 1]])

fc\_y = np.array([[-1, -np.sqrt(2), -1], [0, 0, 0], [1, np.sqrt(2), 1]])

edges\_x = cv2.filter2D(self.gray, cv2.CV\_64F, fc\_x)

edges\_y = cv2.filter2D(self.gray, cv2.CV\_64F, fc\_y)

edges = np.sqrt(np.square(edges\_x) + np.square(edges\_y))

return edges

def detect\_edges\_laplacian(self):

edges = cv2.Laplacian(self.gray, cv2.CV\_64F)

return edges

def detect\_edges\_canny\_deriche(self):

edges = np.zeros(self.gray.size)

for i in range(1, self.gray.size[0]-1):

for j in range(1, self.gray.size[1]-1):

dx = int(self.gray.getpixel((i, j+1))) –

int(self.gray.getpixel((i+1, j)))

dy = int(self.gray.getpixel((i, j+1))) - int(self.gray.getpixel((i, j)))

edges[i,j] = np.sqrt(dx\*dx + dy\*dy)

edges = np.rot90(edges, k=3)

return edges

def detect\_edges\_kirsch(self):

kirsch\_kernels = [

np.array([[-3, -3, 5], [-3, 0, 5], [-3, -3, 5]]),

np.array([[-3, 5, 5], [-3, 0, 5], [-3, -3, -3]]),

np.array([[5, 5, 5], [-3, 0, -3], [-3, -3, -3]]),

np.array([[5, 5, -3], [5, 0, -3], [-3, -3, -3]]),

np.array([[5, -3, -3], [5, 0, -3], [5, -3, -3]]),

np.array([[-3, -3, -3], [5, 0, -3], [5, 5, -3]]),

np.array([[-3, -3, -3], [-3, 0, -3], [5, 5, 5]]),

np.array([[-3, -3, -3], [-3, 0, 5], [-3, 5, 5]])

]

edges = np.zeros(self.gray.size)

for kernel in kirsch\_kernels:

edges += np.absolute(cv2.filter2D(self.gray, cv2.CV\_64F, kernel))

min\_val, max\_val, \_, \_ = cv2.minMaxLoc(edges)

edges = (255 \* ((edges - min\_val) / (max\_val - min\_val)))

return edges

def Marr\_Hildreth\_edge\_detector(self,ksize):

return cv2.Laplacian(self.gray, cv2.CV\_64F, ksize=ksize)

def zero\_crossing\_detect\_edges(self):

gray = cv2.cvtColor(self.img, cv2.COLOR\_BGR2GRAY)

edges = cv2.Canny(self.gray, 100, 200)

\_, contours, \_ = cv2.findContours(edges, cv2.RETR\_TREE, cv2.CHAIN\_APPROX\_NONE)

polygons = []

for contour in contours:

approx = cv2.approxPolyDP(contour, 0.1 \* cv2.arcLength(contour, True), True)

polygons.append(approx)

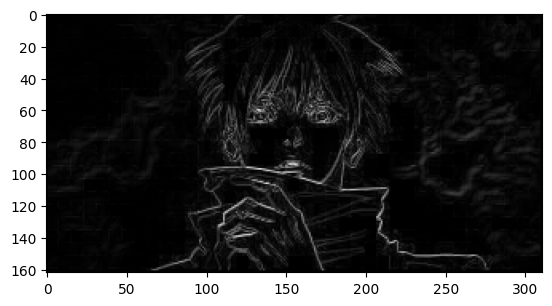
zero\_crossing\_edges = self.img.copy()

cv2.drawContours(zero\_crossing\_edges, polygons, -1, (0, 0, 0), 3)

return zero\_crossing\_edges

**OUTPUT:**

****

****

**CONCLUSION:**

Image edge detection is a fundamental technique in computer vision and image processing. It plays a crucial role in identifying object boundaries and extracting important features from images. Various edge detection algorithms, such as the Sobel, Canny, and Prewitt operators, offer different trade-offs in terms of accuracy and computational complexity. The choice of algorithm depends on the specific application and requirements. Overall, edge detection is a key step in many image

analysis tasks, including object recognition, image segmentation, and more, making it an essential tool in the field of computer vision.