|  |
| --- |
|  |
| **Image Processing Technique**  With the help of python |
| |  |  |  | | --- | --- | --- | | M Ihtesham Khan | 5/23/24 | Image processing | |

Image preprocessing into several steps and provide details for each step with examples:

1. **Image Acquisition:**

Image preprocessing begins with acquiring or capturing the image from a source such as a camera, scanner, or file.

**2. Image Conversion:**

Convert the acquired image to a suitable format for further processing. Common formats include grayscale and RGB.

Example: Converting a color image to grayscale using OpenCV:

python

import cv2

# Read the image

image = cv2.imread('input\_image.jpg')

# Convert to grayscale

gray\_image = cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY)

**3. Image Enhancement:**

Enhance the image quality by adjusting contrast, brightness, or sharpness.

Example: Applying histogram equalization for contrast enhancement using OpenCV:

python

import cv2

# Read the grayscale image

gray\_image = cv2.imread('input\_image.jpg', cv2.IMREAD\_GRAYSCALE)

# Apply histogram equalization

equalized\_image = cv2.equalizeHist(gray\_image)

**4. Noise Reduction:**

Remove or reduce noise in the image to improve clarity and quality.

**Example**: Applying Gaussian blur for noise reduction using OpenCV:

python

import cv2

# Read the image

image = cv2.imread('input\_image.jpg')

# Apply Gaussian blur

blurred\_image = cv2.GaussianBlur(image, (5, 5), 0)

**5.Image Scaling and Resizing:**

Resize the image to a specific size or scale for consistency or to fit processing requirements.

**Example:** Resizing the image to a specific width and height using OpenCV:

python

import cv2

# Read the image

image = cv2.imread('input\_image.jpg')

# Resize the image to a specific width and height

resized\_image = cv2.resize(image, (width, height))

**6. Image Rotation and Transformation:**

Rotate or transform the image to correct orientation or perspective distortions.

**Example:** Rotating the image by a specified angle using OpenCV:

python

import cv2

# Read the image

image = cv2.imread('input\_image.jpg')

# Rotate the image by 90 degrees clockwise

rotated\_image = cv2.rotate(image, cv2.ROTATE\_90\_CLOCKWISE)

**7. Image Cropping:**

Crop a specific region of interest from the image.

Example: Cropping a region of interest from the image using OpenCV:

python

import cv2

# Read the image

image = cv2.imread('input\_image.jpg')

# Define the region of interest (ROI)

x, y, width, height = 100, 100, 200, 200

# Crop the ROI from the image

cropped\_image = image[y:y+height, x:x+width]

**8. Image Normalization**:

Normalize the pixel values of the image to a specific range or distribution.

Example: Normalizing pixel values to the range [0, 1] using scikit-image:

python

from skimage import exposure

# Read the image

image = cv2.imread('input\_image.jpg')

# Convert to grayscale

gray\_image = cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY)

# Normalize pixel values to the range [0, 1]

normalized\_image = exposure.rescale\_intensity(gray\_image)

**9. Image Thresholding:**

Convert the image into a binary image based on a specified threshold value.

Example: Applying global thresholding to convert the image into binary using OpenCV:

python

import cv2

# Read the grayscale image

gray\_image = cv2.imread('input\_image.jpg', cv2.IMREAD\_GRAYSCALE)

# Apply global thresholding

\_, binary\_image = cv2.threshold(gray\_image, 128, 255, cv2.THRESH\_BINARY)

**10. Edge Detection:**

Detect edges in the image to highlight object boundaries.

Example: Detecting edges using the Canny edge detector in OpenCV:

python

import cv2

# Read the grayscale image

gray\_image = cv2.imread('input\_image.jpg', cv2.IMREAD\_GRAYSCALE)

# Apply Canny edge detection

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

**Morphological operation:**

**Morphological operations in** **Image processing using OpenCV**

Morphological operations are used to process images based on shapes. They are most commonly applied to binary images, but they can also be used on grayscale images. The two fundamental operations are *dilation and erosion.* Let's go through each of them:

**1. Dilation:**

Dilation adds pixels to the boundaries of objects in an image. It's useful for joining broken parts of an object, filling holes, and smoothing the boundaries.

**Key Points:**

* Dilation increases the size of objects.
* It's effective for joining broken parts of an object and filling in gaps.
* The size and shape of the structuring element determine the extent of dilation.
* Iterating dilation multiple times leads to more aggressive expansion.

Here's a step-by-step guide with Python code using OpenCV:

Python :

1. Import necessary libraries:

python

import cv2

import numpy as np

#2. Read the image:

image = cv2.imread('input\_image.png', 0) # Read image as grayscale

#3. Define a kernel (structuring element):

kernel = np.ones((5, 5), np.uint8) # Kernel of 5x5 ones

dilated\_image = cv2.dilate(image, kernel, iterations=1)

**2. Erosion:**

Erosion removes pixels at the boundaries of objects in an image. It's useful for removing noise, detaching connected objects, and separating objects from the background.

K**ey Points:**

* Erosion reduces the size of objects.
* It's useful for removing small noise, separating connected objects, and smoothing object boundaries.
* The size and shape of the structuring element determine the extent of erosion.
* Repeated erosion can erode away smaller details.

Let's continue from step 3:

#3. Define a kernel (structuring element) if not already done.

#4. Apply erosion:

python

eroded\_image = cv2.erode(image, kernel, iterations=1)

Example:

Consider a binary image with a white square in the center surrounded by black pixels.

Input Image:

00000

01110

01110

01110

00000

Dilated Image (with a 3x3 kernel):

01110

11111

11111

11111

01110

Eroded Image (with a 3x3 kernel):

00000

01110

01110

01110

00000

These operations can be chained together and used in combination with other image processing techniques for various tasks such as noise removal, object detection, and image segmentation.

**How to reduce Noice from the Image ?**

Reducing noise from an image is a common preprocessing step in image processing, as noise can degrade the quality of the image and interfere with subsequent analysis or processing. Here are some techniques to reduce noise from an image:

**1. Gaussian Blurring:**

Gaussian blurring is a widely used technique to reduce noise while preserving the edges in an image. It works by convolving the image with a Gaussian kernel. This operation effectively smooths the image and reduces high-frequency noise.

python

blurred\_image = cv2.GaussianBlur(image, (5, 5), 0)

**2. Median Filtering:**

Median filtering is effective for removing salt-and-pepper noise, which appears as random, isolated white and black pixels. It replaces each pixel's value with the median value of its neighborhood.

Python ::: filtered\_image = cv2.medianBlur(image, 5)

**3. Bilateral Filtering:**

Bilateral filtering is useful for reducing noise while preserving edges in the image. It applies a Gaussian filter in the spatial domain and a Gaussian filter in the intensity domain simultaneously.

python

filtered\_image = cv2.bilateralFilter(image, 9, 75, 75)

**4. Non-Local Means Denoising:**

Non-local means denoising is a powerful technique for removing Gaussian noise. It works by averaging similar patches within the image, effectively reducing noise while preserving image details.

python

filtered\_image = cv2.fastNlMeansDenoising(image, None, h=10, templateWindowSize=7, searchWindowSize=21)

**5. Wiener Filtering:**

Wiener filtering is effective for reducing additive noise by estimating the original image from the noisy image and the point spread function (PSF) of the noise.

python

filtered\_image = cv2.wiener(image, (5, 5))

**Choosing the Right Filter:**

The choice of filter depends on the type of noise present in the image and the desired trade-off between noise reduction and preservation of image details.

Experiment with different filters and parameters to find the most suitable one for your specific application.

Example:

python

import cv2

# Read the noisy image

noisy\_image = cv2.imread('noisy\_image.png')

# Apply Gaussian blur

blurred\_image = cv2.GaussianBlur(noisy\_image, (5, 5), 0)

# Display the original and filtered images

cv2.imshow('Noisy Image', noisy\_image)

cv2.imshow('Blurred Image', blurred\_image)

cv2.waitKey(0)

cv2.destroyAllWindows()

Additional Notes:

It's important to adjust the parameters of each filter according to the level and type of noise present in the image.

Noise reduction techniques should be applied judiciously to avoid over-smoothing and loss of important image details.

**Image segmentation**

is a process of partitioning an image into multiple segments (sets of pixels, also known as superpixels) to simplify or change the representation of an image into something more meaningful and easier to analyze. Here are detailed explanations of various segmentation techniques with step-by-step instructions and code examples.

**1. Thresholding**

Global Thresholding

Thresholding converts an image into a binary image based on a threshold value.

python

import cv2

import matplotlib.pyplot as plt

# Read the grayscale image

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

# Apply global thresholding

\_, binary\_image = cv2.threshold(image, 128, 255, cv2.THRESH\_BINARY)

# Display the image

plt.imshow(binary\_image, cmap='gray')

plt.title('Global Thresholding')

plt.axis('off')

plt.show()

**Adaptive Thresholding**

Adaptive thresholding calculates the threshold for a small region of the image.

python

import cv2

import matplotlib.pyplot as plt

# Read the grayscale image

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

# Apply adaptive thresholding

adaptive\_image = cv2.adaptiveThreshold(image, 255, cv2.ADAPTIVE\_THRESH\_GAUSSIAN\_C, cv2.THRESH\_BINARY, 11, 2)

# Display the image

plt.imshow(adaptive\_image, cmap='gray')

plt.title('Adaptive Thresholding')

plt.axis('off')

plt.show()

2. **Edge Detection**

Edge detection finds the boundaries of objects within images.

Canny Edge Detection

Canny edge detection uses a multi-stage algorithm to detect edges.

python

import cv2

import matplotlib.pyplot as plt

# Read the grayscale image

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

# Apply Canny edge detection

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

# Display the image

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

plt.title('Canny Edge Detection')

plt.axis('off')

plt.show()

**3. Region-Based Segmentation**

Watershed Algorithm

The watershed algorithm treats grayscale images like topographic surfaces and finds the watershed lines.

python

import cv2

import numpy as np

import matplotlib.pyplot as plt

# Read the image

image = cv2.imread('input\_image.jpg')

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

ret, thresh = cv2.threshold(gray, 0, 255, cv2.THRESH\_BINARY\_INV + cv2.THRESH\_OTSU)

# Remove noise

kernel = np.ones((3, 3), np.uint8)

opening = cv2.morphologyEx(thresh, cv2.MORPH\_OPEN, kernel, iterations=2)

# Sure background area

sure\_bg = cv2.dilate(opening, kernel, iterations=3)

# Finding sure foreground area

dist\_transform = cv2.distanceTransform(opening, cv2.DIST\_L2, 5)

ret, sure\_fg = cv2.threshold(dist\_transform, 0.7 \* dist\_transform.max(), 255, 0)

# Finding unknown region

sure\_fg = np.uint8(sure\_fg)

unknown = cv2.subtract(sure\_bg, sure\_fg)

# Marker labelling

ret, markers = cv2.connectedComponents(sure\_fg)

# Add one to all labels so that sure background is not 0, but 1

markers = markers + 1

# Now, mark the region of unknown with zero

markers[unknown == 0] = 0

# Apply the watershed algorithm

markers = cv2.watershed(image, markers)

image[markers == -1] = [255, 0, 0]

# Display the image

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

plt.title('Watershed Segmentation')

plt.axis('off')

plt.show()

**4. Clustering-Based Segmentation**

K-means Clustering

K-means clustering groups pixels in the feature space into k clusters.

python

import cv2

import numpy as np

import matplotlib.pyplot as plt

# Read the image

image = cv2.imread('input\_image.jpg')

Z = image.reshape((-1, 3))

# Convert to float32

Z = np.float32(Z)

# Define criteria and apply K-means

criteria = (cv2.TERM\_CRITERIA\_EPS + cv2.TERM\_CRITERIA\_MAX\_ITER, 10, 1.0)

K = 8

ret, label, center = cv2.kmeans(Z, K, None, criteria, 10, cv2.KMEANS\_RANDOM\_CENTERS)

# Convert back into uint8 and make original image

center = np.uint8(center)

res = center[label.flatten()]

segmented\_image = res.reshape((image.shape))

# Display the image

plt.imshow(cv2.cvtColor(segmented\_image, cv2.COLOR\_BGR2RGB))

plt.title('K-means Segmentation')

plt.axis('off')

plt.show()

**5. Contour Detection**

Contour detection finds the boundaries of shapes in images.

python

import cv2

import matplotlib.pyplot as plt

# Read the image

image = cv2.imread('input\_image.jpg')

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

ret, thresh = cv2.threshold(gray, 127, 255, cv2.THRESH\_BINARY)

# Find contours

contours, hierarchy = cv2.findContours(thresh, cv2.RETR\_TREE, cv2.CHAIN\_APPROX\_SIMPLE)

# Draw contours

image\_contours = image.copy()

cv2.drawContours(image\_contours, contours, -1, (0, 255, 0), 3)

# Display the image

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

plt.title('Contour Detection')

plt.axis('off')

plt.show()

**6. Superpixel Segmentation**

SLIC (Simple Linear Iterative Clustering)

SLIC segments the image into superpixels.

python

import cv2

import numpy as np

import matplotlib.pyplot as plt

# Read the image

image = cv2.imread('input\_image.jpg')

# Apply SLIC algorithm

slic = cv2.ximgproc.createSuperpixelSLIC(image, algorithm=cv2.ximgproc.SLICO, region\_size=20, ruler=10.0)

slic.iterate(10)

mask\_slic = slic.getLabelContourMask()

label\_slic = slic.getLabels()

# Create mask

mask\_inv\_slic = cv2.bitwise\_not(mask\_slic)

# Apply mask

image\_slic = cv2.bitwise\_and(image, image, mask=mask\_inv\_slic)

# Display the image

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

plt.title('SLIC Superpixel Segmentation')

plt.axis('off')

plt.show()

**7. Deep Learning-Based Segmentation**

U-Net

U-Net is a convolutional network architecture for fast and precise segmentation of images. Due to its complexity, implementing U-Net from scratch requires a deep learning framework like TensorFlow or PyTorch, and a dataset for training. Here's a simplified implementation using Keras:

**python**

import numpy as np

from tensorflow.keras.models import \*

from tensorflow.keras.layers import \*

from tensorflow.keras.optimizers import \*

def unet(input\_size=(256,256,1)):

inputs = Input(input\_size)

conv1 = Conv2D(64, 3, activation='relu', padding='same', kernel\_initializer='he\_normal')(inputs)

conv1 = Conv2D(64, 3, activation='relu', padding='same', kernel\_initializer='he\_normal')(conv1)

pool1 = MaxPooling2D(pool\_size=(2, 2))(conv1)

conv2 = Conv2D(128, 3, activation='relu', padding='same', kernel\_initializer='he\_normal')(pool1)

conv2 = Conv2D(128, 3, activation='relu', padding='same', kernel\_initializer='he\_normal')(conv2)

pool2 = MaxPooling2D(pool\_size=(2, 2))(conv2)

conv3 = Conv2D(256, 3, activation='relu', padding='same', kernel\_initializer='he\_normal')(pool2)

conv3 = Conv2D(256, 3, activation='relu', padding='same', kernel\_initializer='he\_normal')(conv3)

pool3 = MaxPooling2D(pool\_size=(2, 2))(conv3)

conv4 = Conv2D(512, 3, activation='relu', padding='same', kernel\_initializer='he\_normal')(pool3)

conv4 = Conv2D(512, 3, activation='relu', padding='same', kernel\_initializer='he\_normal')(conv4)

drop4 = Dropout(0.5)(conv4)

pool4 = MaxPooling2D(pool\_size=(2, 2))(drop4)

conv5 = Conv2D(1024, 3, activation='relu', padding='same', kernel\_initializer='he\_normal')(pool4)

conv5 = Conv2D(1024, 3, activation='relu', padding='same', kernel\_initializer='he\_normal')(conv5)

drop5 = Dropout(0.5)(conv5)

up6 = Conv2D(512, 2, activation='relu', padding='same', kernel\_initializer='he\_normal')(UpSampling2D(size=(2, 2))(drop5))

merge6 = concatenate([drop4, up6], axis=3)

conv6 =

Conv2D(512, 3, activation='relu', padding='same', kernel\_initializer='he\_normal')(merge6)

conv6 = Conv2D(512, 3, activation='relu', padding='same', kernel\_initializer='he\_normal')(conv6)

up7 = Conv2D(256, 2, activation='relu', padding='same', kernel\_initializer='he\_normal')(UpSampling2D(size=(2, 2))(conv6))

merge7 = concatenate([conv3, up7], axis=3)

conv7 = Conv2D(256, 3, activation='relu', padding='same', kernel\_initializer='he\_normal')(merge7)

conv7 = Conv2D(256, 3, activation='relu', padding='same', kernel\_initializer='he\_normal')(conv7)

up8 = Conv2D(128, 2, activation='relu', padding='same', kernel\_initializer='he\_normal')(UpSampling2D(size=(2, 2))(conv7))

merge8 = concatenate([conv2, up8], axis=3)

conv8 = Conv2D(128, 3, activation='relu', padding='same', kernel\_initializer='he\_normal')(merge8)

conv8 = Conv2D(128, 3, activation='relu', padding='same', kernel\_initializer='he\_normal')(conv8)

up9 = Conv2D(64, 2, activation='relu', padding='same', kernel\_initializer='he\_normal')(UpSampling2D(size=(2, 2))(conv8))

merge9 = concatenate([conv1, up9], axis=3)

conv9 = Conv2D(64, 3, activation='relu', padding='same', kernel\_initializer='he\_normal')(merge9)

conv9 = Conv2D(64, 3, activation='relu', padding='same', kernel\_initializer='he\_normal')(conv9)

conv9 = Conv2D(2, 3, activation='relu', padding='same', kernel\_initializer='he\_normal')(conv9)

conv10 = Conv2D(1, 1, activation='sigmoid')(conv9)

model = Model(inputs=inputs, outputs=conv10)

model.compile(optimizer=Adam(lr=1e-4), loss='binary\_crossentropy', metrics=['accuracy'])

return model

# Assuming you have loaded and preprocessed your data:

# x\_train, y\_train = load\_data()

model = unet()

model.summary()

# Train the model

# model.fit(x\_train, y\_train, batch\_size=32, epochs=100)

The Neuroimaging Informatics Technology Initiative (NIfTI) is a standardized data format used for storing brain imaging data, particularly from MRI (Magnetic Resonance Imaging) scans. It helps researchers and clinicians to share and analyze brain scan data more easily and accurately.

### Key Points:

1. \*\*Standardized Format\*\*: NIfTI provides a consistent way to store brain imaging data, ensuring that files can be easily read and processed by different software programs.

2. \*\*File Structure\*\*: NIfTI files typically contain two parts:

- A header that includes information about the data (like dimensions and voxel size).

- The image data itself, usually in a compressed format.

3. \*\*Enhanced Analysis\*\*: By using a common format, researchers can more effectively compare and combine data from different studies, improving the reliability of neuroimaging research.

### Example in Simple Terms:

Imagine you have a photo of a brain taken by an MRI machine. The NIfTI format is like saving that photo as a high-quality, standard JPEG file that can be opened and understood by any image viewer. This makes it easy to share the photo with other doctors or researchers, who can then use their own software to look at and analyze the photo in detail.

### Practical Use:

For instance, if a research team in the US collects brain scans from patients with Alzheimer's and saves them in NIfTI format, a team in Europe can easily access these scans, analyze them using their own tools, and contribute to a larger study without worrying about compatibility issues.

In summary, NIfTI is a vital tool in brain imaging research, making data sharing and analysis more efficient and standardized.

**Some references where I collect the data**

**Digital Imaging and Communications in Medicine**

[encord.com/blog/improve-medical-imaging-dataset-machine-learning/](https://encord.com/blog/improve-medical-imaging-dataset-machine-learning/)