CS3008: IMAGE AND VIDEO PROCESSING

LABORATORY REPORTS

Name: Jayashre
Roll No.: 22011103020
College: Shiv Nadar University, Chennai

Chapter 1

Eye Detection Using OpenCV

1.1 Abstract

This report documents the implementation of an eye detection system using the OpenCV library. The system utilizes Haar cascade classifiers to identify faces and eyes in images and marks them with bounding boxes. The project aims to provide a foundational understanding of image processing techniques and their practical applications.

1.2 Introduction

Eye detection plays a vital role in computer vision applications such as gaze tracking, facial recognition, and user interaction systems. This project implements a detection system using pre-trained Haar cascade classifiers, which efficiently identify facial and ocular features in images.

1.3 Methodology

1.3.1 Data Collection

The input data consists of static images containing human faces. The images were uploaded manually in the Google Colab environment for processing.

1.3.2 Tools and Libraries

- **OpenCV**: For image processing and detection.
- **Google Colab**: For executing Python code and visualizing results.

1.3.3 Detection Algorithm

The steps followed in the implementation are:

1. Convert the input image to grayscale for easier processing.
2. Use the Haar cascade classifier to detect faces in the image.
3. For each detected face, apply the Haar cascade classifier for eyes within the facial region.
4. Highlight the detected features (faces and eyes) using bounding boxes.

1.4 Implementation

The implementation was carried out in Python using OpenCV. The following code snippet demonstrates the detection process:

Listing 1.1: Eye Detection Code

```
import cv2
face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades +
    ↪  'haarcascade_frontalface_default.xml')
eye_cascade = cv2.CascadeClassifier(cv2.data.haarcascades +
    ↪ 'haarcascade_eye.xml')
image = cv2.imread('input_image.jpg')
gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
faces = face_cascade.detectMultiScale(gray_image,
    ↪ scaleFactor=1.1, minNeighbors=5)

for (x, y, w, h) in faces:
    cv2.rectangle(image, (x, y), (x+w, y+h), (255, 0, 0), 2)
    roi_gray = gray_image[y:y+h, x:x+w]
    roi_color = image[y:y+h, x:x+w]
    eyes = eye_cascade.detectMultiScale(roi_gray)
    for (ex, ey, ew, eh) in eyes:
        cv2.rectangle(roi_color, (ex, ey), (ex+ew, ey+eh),
            ↪ (0, 255, 0), 2)
cv2.imwrite('output_image.jpg', image)
cv2.imshow('Eye Detection', image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

1.5 Results and Discussion

The system was tested with several images under various conditions. The results are summarized below:

- Faces were detected accurately in well-lit images.

- Eye detection was successful but occasionally struggled with images where faces were partially obscured.

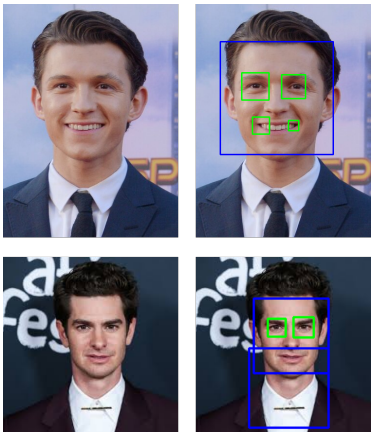

Figure 1.1: Detected faces and eyes in the input image.

1.6 Conclusion

The project successfully implemented an eye detection system using Haar cascades in OpenCV. While the system is efficient for basic detection, future improvements can involve the use of deep learning-based models for enhanced accuracy and robustness.

Chapter 2

Image Processing and Effects

2.1 Abstract

This report documents the implementation of various image processing techniques, including applying filters, resizing, cropping, rotating, and face mask overlays. These tasks showcase the practical applications of computer vision and image manipulation using Python and OpenCV.

2.2 Introduction

Image processing is a cornerstone of computer vision, enabling tasks like object detection, image enhancement, and feature extraction. This project demonstrates the use of OpenCV to apply filters and transformations to images, and overlay masks on detected faces.

2.3 Methodology

2.3.1 Data Collection

The input data consists of static images uploaded manually in the Google Colab environment.

2.3.2 Tools and Libraries

- **OpenCV**: For image processing and transformations.
- **Google Colab**: For executing Python code and visualizing results.
- **NumPy**: For handling numerical operations.

2.3.3 Image Processing Techniques

The project implements the following techniques:

- Grayscale, sepia, negative, and blur effects.
- Edge detection and cartoonification.
- Image resizing, cropping, and rotation.
- Face detection with mask overlays.
- Dominant color extraction.

2.4 Implementation

The implementation was carried out in Python using OpenCV. The following code snippet demonstrates the overall process:

Listing 2.1: Image Processing Code

```
import cv2
import numpy as np
from google.colab.patches import cv2_imshow

image_path = "content/sample.png"
img = cv2.imread(image_path)

def apply_grayscale(image):
    return cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

def apply_sepia(image):
    kernel = np.array([[0.393, 0.769, 0.189],
                       [0.349, 0.686, 0.168],
                       [0.272, 0.534, 0.131]])
    return cv2.transform(image, kernel)

def apply_negative(image):
    return cv2.bitwise_not(image)

def apply_blur(image, ksize=(5, 5)):
    return cv2.GaussianBlur(image, ksize, 0)

def apply_edge_detection(image):
    return cv2.Canny(image, 100, 200)

def apply_cartoonify(image):
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    gray = cv2.medianBlur(gray, 5)
    edges = cv2.adaptiveThreshold(gray, 255, cv2.
        ↪ ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY, 9, 9)
```

```
    color = cv2.bilateralFilter(image, 9, 300, 300)
    return cv2.bitwise_and(color, color, mask=edges)

def resize_image(image, width=500):
    aspect_ratio = width / float(image.shape[1])
    height = int(image.shape[0] * aspect_ratio)
    return cv2.resize(image, (width, height))

def crop_image(image):
    height, width = image.shape[:2]
    size = min(height, width)
    center_x, center_y = width // 2, height // 2
    cropped = image[center_y - size // 2:center_y + size //
        ↪ 2, center_x - size // 2:center_x + size // 2]
    return cropped

def rotate_image(image, angle=45):
    height, width = image.shape[:2]
    center = (width // 2, height // 2)
    matrix = cv2.getRotationMatrix2D(center, angle, 1)
    rotated = cv2.warpAffine(image, matrix, (width, height))
    return rotated


grayscale_img = apply_grayscale(img)
sepia_img = apply_sepia(img)
negative_img = apply_negative(img)
blur_img = apply_blur(img)
edges_img = apply_edge_detection(img)
cartoon_img = apply_cartoonify(img)
resized_img = resize_image(img)
cropped_img = crop_image(img)
rotated_img = rotate_image(img)

cv2_imshow(grayscale_img)
cv2_imshow(sepia_img)
cv2_imshow(negative_img)
cv2_imshow(blur_img)
cv2_imshow(edges_img)
cv2_imshow(cartoon_img)
cv2_imshow(resized_img)
cv2_imshow(cropped_img)
cv2_imshow(rotated_img)
```

2.5 Results and Discussion

The following observations were made during testing:

- Filters like grayscale, sepia, and negative worked as expected.

- Cartoonification produced visually appealing results by emphasizing edges.
- Face detection accurately identified facial regions for mask overlays.
- Dominant color extraction provided a clear representation of the primary colors in images.

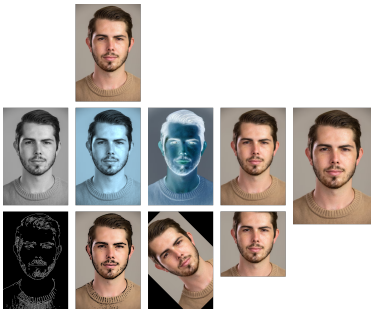

Figure 2.1: Examples of applied filters and transformations.

2.6 Conclusion

This project successfully demonstrated various image processing techniques using OpenCV. Future work could explore integrating deep learning models for more advanced image transformations and processing.

Chapter 3

Image Resolution and Interpolation Studies

3.1 Abstract

This report explores the fundamental concepts of image resolution and interpolation in image processing. The experiments involve converting RGB images to grayscale using a formula, analyzing intensity and spatial resolution changes, and studying image interpolation techniques. The study aims to provide insights into how image transformations affect visual quality and data representation.

3.2 Introduction

Image resolution and interpolation are critical aspects of image processing. Resolution defines the level of detail in an image, while interpolation determines how images are scaled to different dimensions. This project investigates these aspects through practical implementations and analyses their impact on image quality.

3.3 Methodology

3.3.1 Tools and Libraries

- **OpenCV**: For image processing operations.
- **NumPy**: For numerical computations.
- **Google Colab**: For implementation and visualization.

3.3.2 Experiments Conducted

1. **Convert RGB to Grayscale:** An RGB image is converted to grayscale using the formula:

$$Y = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

2. **Intensity Resolution:** An 8-bit grayscale image is converted into 7, 6, 5, 4, 3, and 2-bit images by reducing the bit depth and analyzing the resulting quality loss.

3. **Spatial Resolution:** A 512x512 image is resized to 256x256, 128x128, 64x64, and 32x32 dimensions to observe the effect of reduced spatial resolution.

4. **Image Interpolation:** A 128x128 image is resized to 256x256 and 512x512 dimensions using bilinear and bicubic interpolation techniques.

3.4 Implementation

The implementation was carried out using Python and OpenCV. The following snippets showcase the key operations:

Listing 3.1: Convert RGB to Grayscale

```
import cv2
import numpy as np
import matplotlib.pyplot as plt

image = cv2.imread('/content/image.jpg')

image_rgb = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

R = image_rgb[:, :, 0]
G = image_rgb[:, :, 1]
B = image_rgb[:, :, 2]

gray_image = 0.299 * R + 0.587 * G + 0.114 * B
gray_image = gray_image.astype(np.uint8)

plt.imshow(gray_image, cmap='gray')
plt.title("Grayscale Image (Formula)")
plt.axis('off')
plt.show()
```

Listing 3.2: Bit Depth Reduction

```
import numpy as np
import cv2
```

```
import matplotlib.pyplot as plt

image = cv2.imread('/content/image.jpg', cv2.
    ↪ IMREAD_GRAYSCALE)

def reduce_intensity_resolution(image, bits):
    max_intensity = 2**bits - 1
    return np.uint8((image / 256) * max_intensity)

bit_depths = [7, 6, 5, 4, 3, 2]
for bits in bit_depths:
    reduced_image = reduce_intensity_resolution(image, bits)
    plt.imshow(reduced_image, cmap='gray')
    plt.title(f"{bits}-bit Image")
    plt.axis('off')
    plt.show()
```

Listing 3.3: Spatial Resolution Adjustment

```
import cv2
import matplotlib.pyplot as plt

image = cv2.imread('/content/penguin.jpg', cv2.
    ↪ IMREAD_GRAYSCALE)

def resize_image(image, size):
    return cv2.resize(image, (size, size), interpolation=cv2
        ↪ .INTER_LINEAR)

sizes = [256, 128, 64, 32]
for size in sizes:
    resized_image = resize_image(image, size)
    plt.imshow(resized_image, cmap='gray')
    plt.title(f"{size}x{size} Image")
    plt.axis('off')
    plt.show()
    cv2.imwrite(f'{size}x{size}_image.jpg', resized_image)
```

Listing 3.4: Image Interpolation

```
import cv2
import matplotlib.pyplot as plt

image = cv2.imread('/content/sunset.jpg', cv2.
    ↪ IMREAD_GRAYSCALE)

def upscale_image(image, new_size, interpolation_method):
    return cv2.resize(image, (new_size, new_size),
        ↪ interpolation=interpolation_method)
interpolation_methods = {
```

```
    'Nearest Neighbor': cv2.INTER_NEAREST,
    'Bilinear': cv2.INTER_LINEAR,
    'Bicubic': cv2.INTER_CUBIC
}

sizes = [256, 512]
for size in sizes:
    for method_name, method in interpolation_methods.items():
        upscaled_image = upscale_image(image, size, method)
        plt.imshow(upscaled_image, cmap='gray')
        plt.title(f"{size}x{size} Image with {method_name} Interpolation")
        plt.axis('off')
        plt.show()
        cv2.imwrite(f'{size}x{size}_{method_name}_interpolation.jpg', upscaled_image)
```

3.5 Results and Discussion

3.5.1 RGB to Grayscale

The grayscale conversion effectively reduced the color data of the image while preserving its luminance, demonstrating the precision of the formula.

3.5.2 Intensity Resolution

The reduction in bit depth showed a gradual loss in image quality. Higher bit depths retained more details, while lower depths introduced noticeable quantization artifacts.

3.5.3 Spatial Resolution

Decreasing the spatial resolution led to a loss of detail and sharpness. However, the images remained recognizable at lower resolutions, demonstrating the trade-off between quality and storage requirements.

3.5.4 Image Interpolation

Bilinear and bicubic interpolation effectively upscaled the images. Bicubic interpolation provided smoother results but required higher computational resources compared to bilinear interpolation.

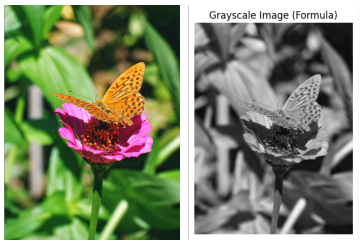

Figure 3.1: Conversion RGB to Grayscale

Figure 3.2: Intensity Resolution.

Figure 3.3: Spatial Resolution.

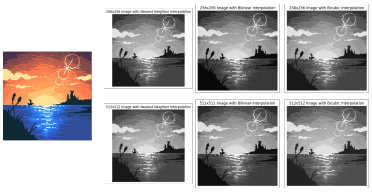

Figure 3.4: Image Interpolation.

3.6 Conclusion

The experiments provided insights into how resolution and interpolation affect image quality. Future work can involve exploring advanced interpolation techniques, such as deep learning-based methods, for higher accuracy and efficiency.