OpenCV Notes
Computer Vision Fundamentals

Course Notes

July 30, 2025

Contents

1 Introduction to OpenCV

1.1 What is OpenCV?

OpenCV (Open Source Computer Vision Library) is an open-source library that includes several hundreds of computer vision algorithms. It was originally developed by Intel and is now maintained by a community of developers.

1.2 Key Features

- Cross-platform compatibility (Windows, Linux, macOS, Android, iOS)
- Supports multiple programming languages (Python, C++, Java)
- Extensive collection of image processing and computer vision algorithms
- Real-time image processing capabilities
- Machine learning algorithms integration

1.3 Installation

To install OpenCV for Python:

```
pip install opencv-python
pip install opencv-contrib-python  # For additional modules
```

1.4 Basic Import and Setup

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

# Basic image reading
img = cv2.imread('image.jpg')
cv2.imshow('Image', img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

2 Image Basics and Color Spaces

2.1 Image Representation

In OpenCV, images are represented as NumPy arrays:

- Grayscale images: 2D arrays (height $\times$ width)
- Color images: 3D arrays (height $\times$ width $\times$ channels)
- Default color format: BGR (Blue, Green, Red)

2.2 Converting to Grayscale

2.2.1 Theory

Grayscale conversion reduces a color image to shades of gray by combining the RGB channels. The most common formula uses weighted averages:

$$\text{Gray} = 0.299 \times R + 0.587 \times G + 0.114 \times B$$

This weighting accounts for human eye sensitivity to different colors.

2.2.2 Implementation

```python
# Method 1: Using cv2.cvtColor()
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

# Method 2: Reading directly as grayscale
gray = cv2.imread('image.jpg', cv2.IMREAD_GRAYSCALE)

# Method 3: Manual conversion
gray = 0.299 * img[:,:,2] + 0.587 * img[:,:,1] + 0.114 * img[:,:,0]
```

2.2.3 Applications

- Reduces computational complexity
- Preprocessing for many computer vision algorithms
- Memory efficiency
- Noise reduction in some cases

3 Image Resizing

3.1 Theory

Image resizing involves changing the dimensions of an image. This requires interpolation methods to determine pixel values in the new image.

3.2 Interpolation Methods

- **Nearest Neighbor (cv2.INTER_NEAREST)**: Fastest but lowest quality
- **Bilinear (cv2.INTER_LINEAR)**: Good balance of speed and quality
- **Bicubic (cv2.INTER_CUBIC)**: Higher quality, slower
- **Lanczos (cv2.INTER_LANCZOS4)**: Highest quality, slowest

3.3 Implementation

```python
# Resize to specific dimensions
resized = cv2.resize(img, (width, height), interpolation=cv2.
    INTER_LINEAR)

# Resize by scale factor
scale_percent = 50  # percent of original size
width = int(img.shape[1] * scale_percent / 100)
height = int(img.shape[0] * scale_percent / 100)
resized = cv2.resize(img, (width, height), interpolation=cv2.INTER_AREA)

# Maintain aspect ratio
def resize_with_aspect_ratio(image, width=None, height=None, inter=cv2.
    INTER_AREA):
    dim = None
    (h, w) = image.shape[:2]

    if width is None and height is None:
        return image
    if width is None:
        r = height / float(h)
        dim = (int(w * r), height)
    else:
        r = width / float(w)
        dim = (width, int(h * r))

    return cv2.resize(image, dim, interpolation=inter)
```

4 Image Blurring

4.1 Theory

Blurring reduces image noise and detail by averaging pixel values with their neighbors. It's implemented using convolution with various kernels.

4.2 Types of Blurring

4.2.1 Gaussian Blur

Uses a Gaussian kernel for smooth blurring:

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

```python
# Gaussian blur
blurred = cv2.GaussianBlur(img, (15, 15), 0)
# (15,15) is kernel size, 0 lets OpenCV calculate sigma
```

4.2.2 Average Blur

Simple averaging of neighboring pixels:

```python
# Average blur
blurred = cv2.blur(img, (15, 15))
```

4.2.3 Median Blur

Replaces each pixel with the median of neighboring pixels:

```python
# Median blur (good for salt-and-pepper noise)
blurred = cv2.medianBlur(img, 15)
```

4.2.4 Bilateral Filter

Preserves edges while reducing noise:

```python
# Bilateral filter
# cv2.bilateralFilter(img, d, sigmaColor, sigmaSpace)
blurred = cv2.bilateralFilter(img, 9, 75, 75)
```

4.3 Applications

- Noise reduction
- Preprocessing for edge detection
- Creating artistic effects
- Background separation

5 Edge Detection

5.1 Theory

Edge detection identifies points in an image where brightness changes sharply. Edges correspond to boundaries between objects, shadows, or texture changes.

5.2 Gradient-Based Methods

5.2.1 Sobel Operator

Computes gradient in x and y directions:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * I \tag{1}$$

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * I \tag{2}$$

```python
# Sobel edge detection
sobelx = cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize=5)
sobely = cv2.Sobel(gray, cv2.CV_64F, 0, 1, ksize=5)
sobel_combined = np.sqrt(sobelx**2 + sobely**2)
```

5.2.2 Laplacian

Second derivative operator:

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

```python
# Laplacian edge detection
laplacian = cv2.Laplacian(gray, cv2.CV_64F)
```

5.3 Canny Edge Detection

5.3.1 Theory

The Canny edge detector is a multi-step algorithm:

1. Gaussian smoothing to reduce noise
2. Gradient calculation using Sobel
3. Non-maximum suppression
4. Double thresholding
5. Edge tracking by hysteresis

5.3.2 Implementation

```python
# Canny edge detection
edges = cv2.Canny(gray, threshold1=50, threshold2=150, apertureSize=3)

# With preprocessing
blurred = cv2.GaussianBlur(gray, (5, 5), 0)
edges = cv2.Canny(blurred, 50, 150)
```

5.3.3 Parameter Tuning

- **threshold1**: Lower threshold for edge linking
- **threshold2**: Upper threshold for edge detection
- **apertureSize**: Sobel kernel size (3, 5, or 7)
- Rule of thumb: threshold2 = 2-3 $\times$ threshold1

6 Face Detection using Haar Cascades

6.1 Theory

Haar cascades use machine learning to detect objects based on Haar-like features. The algorithm:

1. Uses integral images for fast feature computation

2. Applies cascade of classifiers
3. Each stage eliminates non-face regions quickly
4. Only promising regions proceed to next stage

6.2 Haar-like Features

Simple rectangular features that capture:

- Edge features
- Line features
- Four-rectangle features

6.3 Implementation

6.3.1 Loading Cascade Classifier

```python
# Load pre-trained cascade
face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades +
                                      'haarcascade_frontalface_default.xml'
    )

# Other available cascades:
# haarcascade_eye.xml
# haarcascade_smile.xml
# haarcascade_profileface.xml
```

6.3.2 Face Detection

```python
def detect_faces(img):
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # Detect faces
    faces = face_cascade.detectMultiScale(
        gray,
        scaleFactor=1.1,        # How much image size is reduced at each
     scale
        minNeighbors=5,         # How many neighbors each face should
    have
        minSize=(30, 30),       # Minimum possible face size
        flags=cv2.CASCADE_SCALE_IMAGE
    )

    # Draw rectangles around faces
    for (x, y, w, h) in faces:
        cv2.rectangle(img, (x, y), (x+w, y+h), (255, 0, 0), 2)

    return img, faces

# Usage
result_img, detected_faces = detect_faces(img)
print(f"Found {len(detected_faces)} faces")
```

6.3.3 Real-time Face Detection

```python
def real_time_face_detection():
    cap = cv2.VideoCapture(0)  # Use webcam

    while True:
        ret, frame = cap.read()
        if not ret:
            break

        # Detect faces
        result_frame, faces = detect_faces(frame)

        # Display result
        cv2.imshow('Face Detection', result_frame)

        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

    cap.release()
    cv2.destroyAllWindows()
```

6.4 Parameter Optimization

- **scaleFactor**: 1.05-1.3 (smaller = more thorough, slower)
- **minNeighbors**: 3-6 (higher = fewer false positives)
- **minSize**: Depends on expected face size in image
- **maxSize**: Optional upper limit for face size

7 Practical Examples and Complete Programs

7.1 Image Processing Pipeline

```python
import cv2
import numpy as np

def image_processing_pipeline(image_path):
    # Read image
    img = cv2.imread(image_path)
    original = img.copy()

    # Convert to grayscale
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    # Resize image
    height, width = gray.shape
    new_width = 800
    new_height = int(height * new_width / width)
    resized = cv2.resize(gray, (new_width, new_height))

    # Apply Gaussian blur
    blurred = cv2.GaussianBlur(resized, (5, 5), 0)

    # Edge detection
    edges = cv2.Canny(blurred, 50, 150)

    # Face detection
    face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades +
                                         'haarcascade_frontalface_default.
   xml')
    faces = face_cascade.detectMultiScale(resized, 1.1, 4)

    # Draw face rectangles
    face_img = cv2.cvtColor(resized, cv2.COLOR_GRAY2BGR)
    for (x, y, w, h) in faces:
        cv2.rectangle(face_img, (x, y), (x+w, y+h), (255, 0, 0), 2)

    return {
        'original': original,
        'grayscale': resized,
        'blurred': blurred,
        'edges': edges,
        'faces': face_img,
        'face_count': len(faces)
    }

# Usage example
results = image_processing_pipeline('sample_image.jpg')
print(f"Detected {results['face_count']} faces")
```

7.2 Webcam Application

```python
def webcam_processing():
    cap = cv2.VideoCapture(0)
    face_cascade = cv2.CascadeClassifier(cv2.data.haarcascades +
                                         'haarcascade_frontalface_default.
   xml')

    while True:
        ret, frame = cap.read()
        if not ret:
            break

        # Convert to grayscale
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

        # Apply blur
        blurred = cv2.GaussianBlur(gray, (5, 5), 0)

        # Edge detection
        edges = cv2.Canny(blurred, 50, 150)

        # Face detection
        faces = face_cascade.detectMultiScale(gray, 1.1, 4)

        # Draw rectangles around faces
        for (x, y, w, h) in faces:
            cv2.rectangle(frame, (x, y), (x+w, y+h), (255, 0, 0), 2)

        # Display windows
        cv2.imshow('Original', frame)
        cv2.imshow('Edges', edges)

        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

    cap.release()
    cv2.destroyAllWindows()
```

8 Best Practices and Tips

8.1 Performance Optimization

- Convert to grayscale early in pipeline
- Resize images to appropriate size
- Use appropriate data types (CV_8U for display, CV_64F for calculations)
- Cache cascade classifiers
- Use ROI (Region of Interest) when possible

8.2 Common Pitfalls

- Forgetting to handle different image formats
- Not checking if image loaded successfully
- Using wrong color space (BGR vs RGB)
- Incorrect parameter values for algorithms
- Memory leaks from not releasing resources

8.3 Debugging Tips

- Always check image dimensions and data types
- Visualize intermediate results
- Use proper error handling
- Test with various image sizes and qualities
- Validate parameter ranges

9 Conclusion

This document covers the fundamental concepts in OpenCV including image basics, color space conversion, resizing, blurring, edge detection, and face detection using Haar cascades. These techniques form the foundation for more advanced computer vision applications.

Key takeaways:

- Understanding image representation is crucial
- Preprocessing (grayscale, resize, blur) improves algorithm performance
- Parameter tuning is essential for optimal results
- Combining multiple techniques creates powerful applications
- Practice with real-time applications enhances understanding

10 References and Further Reading

- OpenCV Documentation: `https://docs.opencv.org/`
- Viola-Jones Face Detection Paper (2001)
- Canny Edge Detection Paper (1986)
- Digital Image Processing by Gonzalez & Woods