National University of Computer & Emerging Sciences, Karachi
Artificial Intelligence-School of Computing
Fall 2024, Lab Manual - 04

Course Code (AI4002)	Course: Computer Vision Lab
Instructor(s):	Sohail Ahmed

Objectives:

1. Introduction to Feature Extraction
2. Common Feature Extraction Techniques
3. Filtering and Convolution
4. Introduction to Edge Detection
5. Common Edge Detection Techniques
6. Corner Detection & its Common Techniques

Introduction to Feature Extraction

Feature extraction is a fundamental technique in computer vision, a field of artificial intelligence that focuses on enabling computers to interpret and understand visual information from the world. In computer vision, feature extraction refers to the process of selecting and transforming relevant information or patterns from raw visual data, such as images or videos, to create a more compact and meaningful representation that can be used for various tasks, such as object recognition, image classification, facial recognition, and more.

- **Raw Visual Data:** Computer vision tasks begin with the acquisition of raw visual data, typically in the form of images or video frames. These data contain a vast amount of information, including pixel values, colors, and textures.
- **Need for Feature Extraction:** Raw visual data is often too complex and high-dimensional for direct analysis and interpretation by algorithms. Feature extraction is essential because it reduces the dimensionality of the data, enhances relevant information, and simplifies the computational requirements of subsequent tasks.
- **Features:** Features are distinctive characteristics or attributes extracted from the raw data. These features should capture essential information about the objects, patterns, or structures present in the images while discarding irrelevant details.

Types of Features:

- **Local Features:** These are extracted from specific regions of an image, such as keypoints, corners, or small patches. Local features are often used for tasks like image matching and object detection.
- **Global Features:** These are computed over the entire image and capture information about its overall characteristics. Examples include color histograms, texture descriptors, and image moments.
- **Feature Extraction Techniques:**
- **Histogram-based methods**: Histograms of color or texture distributions can capture statistical information about the image.
- **Filter-based methods:** Filters like Gabor filters or Haar wavelets can be applied to identify edges, textures, or specific patterns.

- **Deep Learning Features**: Convolutional Neural Networks (CNNs) have revolutionized feature extraction in computer vision. Features are learned directly from the data through the network's layers, and these learned representations are often highly effective for various tasks.

Applications

- **Object Recognition:** Features help identify and classify objects within images or video frames.
- **Face Recognition:** Extracting facial features like eyes, nose, and mouth for identity verification.
- **Gesture Recognition:** Capturing hand or body movements for human-computer interaction.
- **Medical Imaging:** Identifying anomalies or patterns in medical images like X-rays or MRIs.
- **Autonomous Vehicles:** Extracting features from sensor data to detect obstacles and navigate.

Challenges: Feature extraction can be challenging due to variations in lighting, scale, orientation, and noise in real world data. Choosing appropriate feature extraction techniques and parameters is critical for successful computer vision applications.

Feature Matching and Learning: Extracted features are often used in combination with matching or learning algorithms to perform tasks like object recognition or image classification.

Common Feature Extraction Techniques

1. **HOG (Histogram of Oriented Gradients)**

Histogram of Oriented Gradients (HOG) is a widely used feature extraction technique in computer vision and image processing. It is particularly useful for object detection and pedestrian detection tasks. HOG works by capturing information about the local gradient or edge patterns in an image.

Explanation:

HOG focuses on capturing information about the distribution of gradient directions in an image.
It divides the image into small cells, calculates the gradient magnitude and orientation for each pixel within these cells, and then creates histograms of gradient orientations for these cells.
These histograms are then concatenated to form a feature vector that represents the image.

Working Mechanism

- **Image Preprocessing:**
  Convert the input image to grayscale. This simplifies the calculation of gradients.
- **Gradient Computation:**
  Calculate the gradient magnitude and orientation for each pixel in the image. Typically, the Sobel operator is used for this purpose.
- **Cell Division:**
  Divide the image into small cells (e.g., 8x8 pixels) that do not overlap.
- **Histograms of Oriented Gradients:**
  For each cell, create a histogram of gradient orientations.
  The histogram bins represent different orientation ranges (e.g., 0-20 degrees, 20-40 degrees, etc.).
  The histogram values correspond to the frequency of gradient orientations in that cell.
- **Block Normalization:**

Group neighboring cells into blocks (e.g., 2x2 cells).
Normalize the histograms within each block to reduce the effects of lighting variations.

- **Feature Vector:**
  Concatenate all the normalized histograms from the blocks to form the final HOG feature vector.

```
from skimage.feature import hog
from skimage import exposure
import cv2
import matplotlib.pyplot as plt

# Read an image (grayscale)
image = cv2.imread('image.jpg', cv2.IMREAD_GRAYSCALE)

# Compute HOG features
features, hog_image = hog(image, pixels_per_cell=(8, 8), cells_per_block=(2, 2), visualize=True)

# Rescale the HOG image for better visualization
hog_image_rescaled = exposure.rescale_intensity(hog_image, in_range=(0, 10))

# Plot the HOG image and its corresponding feature vector
plt.figure(figsize=(12, 6))
plt.subplot(121)
plt.imshow(image, cmap=plt.cm.gray)
plt.title('Original Image')

plt.subplot(122)
plt.imshow(hog_image_rescaled, cmap=plt.cm.gray)
plt.title('HOG Features')
plt.show()

# Print the HOG feature vector
print("HOG Feature Vector:")
print(features)
```

2. Local Binary Pattern (LBP)

Local Binary Pattern (LBP) is a texture feature extraction technique commonly used in computer vision and image analysis. LBP captures local patterns by comparing the intensity of a pixel with its neighboring pixels. It is particularly useful for texture classification and face recognition.

Explanation:

- LBP operates on grayscale images.
- It defines a circular neighborhood around each pixel and encodes the local texture pattern based on the intensity comparisons within this neighborhood.
- For each pixel in the image, LBP computes a binary pattern by comparing the pixel's intensity with its neighbors. The result is a binary number.

- These binary patterns are collected to form a histogram that represents the texture pattern distribution in the image.
- The histogram of LBP patterns serves as a feature vector for texture analysis.

Working Mechanism

1. **Image Preprocessing:**
   Convert the input image to grayscale if it is not already.
2. **Local Neighborhood Definition:**
   Select a circular neighborhood of a certain radius around each pixel. The neighborhood is defined by its radius and the number of neighboring pixels.
3. **Intensity Comparison:**
   For each pixel in the neighborhood, compare its intensity with the center pixel's intensity. If the neighbor's intensity is greater or equal, assign it a value of 1; otherwise, assign 0.
4. **Pattern Formation:**
   Concatenate the binary values from the comparisons to form an LBP pattern.
5. **Histogram Calculation:**
   Count the occurrences of each LBP pattern in the entire image to create an LBP histogram.
6. **Feature Vector:**
   The LBP histogram serves as the feature vector for the image, which can be used for further analysis or classification.

```
from skimage import feature
import cv2
import numpy as np
import matplotlib.pyplot as plt

# Read an image (grayscale)
image = cv2.imread('image.jpg', cv2.IMREAD_GRAYSCALE)

# Compute LBP features
radius = 1  # Radius of the circular neighborhood
n_points = 8 * radius  # Number of neighboring pixels to consider
lbp_image = feature.local_binary_pattern(image, n_points, radius, method='uniform')

# Calculate the LBP histogram
hist, _ = np.histogram(lbp_image.ravel(), bins=np.arange(0, n_points + 3), range=(0, n_points + 2))

# Normalize the histogram
hist = hist.astype("float")
hist /= (hist.sum() + 1e-6)

# Display the LBP image and histogram
plt.figure(figsize=(12, 6))
plt.subplot(121)
plt.imshow(lbp_image, cmap='gray')
```

```
plt.title('LBP Image')

plt.subplot(122)
plt.bar(range(0, n_points + 2), hist)
plt.title('LBP Histogram')
plt.xlabel('LBP Patterns')
plt.ylabel('Frequency')
plt.show()

# Print the LBP feature vector
print("LBP Feature Vector:")
print(hist)
```

3. **Histogram of Color:**

   Explanation: Captures the distribution of color intensities in an image.

   Working Mechanism: Divides the color space into bins (e.g., RGB or HSV) and counts the number of pixels in each bin.

   Task*: Python Code (Provide the Code)*

4. **Histogram of Edge Directions (HED)**

   Histogram of Edge Directions (HED) is a feature extraction technique that captures the distribution of edge orientations in an image. It provides valuable information about the dominant edge directions, which can be useful for tasks such as texture analysis, object recognition, and image segmentation. Below, I'll explain the working mechanism of HED and provide a Python code example using the OpenCV library.

   Working Mechanism:

   1. **Image Preprocessing:**
      Convert the input image to grayscale if it is not already in grayscale.
      Optionally, you can apply Gaussian smoothing to the image to reduce noise and enhance the edge detection.
   2. **Edge Detection:**
      Apply an edge detection algorithm to the preprocessed image. Common edge detection methods include the Canny edge detector, Sobel operator, or Scharr operator.
      The result is an edge map, where pixels with strong edges are highlighted.
   3. **Orientation Calculation:**
      For each pixel in the edge map, calculate the orientation (angle) of the gradient at that pixel. This can be done using the arctangent function.
   4. **Orientation Binning:**
      Divide the range of orientations into bins (e.g., 360 degrees divided into 8 bins for octagonal directions).
      Assign each pixel's orientation to one of the bins based on its angle.
   5. **Histogram Formation:**
      Count the number of pixels in each orientation bin.
      The resulting histogram represents the distribution of edge directions in the image.
   6. **Feature Vector:**
      The histogram of edge directions serves as the feature vector for the image.

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

# Read an image (grayscale)
image = cv2.imread('image.jpg', cv2.IMREAD_GRAYSCALE)

# Apply Gaussian smoothing to reduce noise (optional)
image_smoothed = cv2.GaussianBlur(image, (5, 5), 0)

# Apply Canny edge detection
edges = cv2.Canny(image_smoothed, threshold1=30, threshold2=70)

# Calculate gradient direction (orientation)
gradient_x = cv2.Sobel(image_smoothed, cv2.CV_64F, 1, 0, ksize=3)
gradient_y = cv2.Sobel(image_smoothed, cv2.CV_64F, 0, 1, ksize=3)
gradient_magnitude = np.sqrt(gradient_x**2 + gradient_y**2)
gradient_orientation = np.arctan2(gradient_y, gradient_x) * 180 / np.pi

# Create a histogram of edge directions
hist, bin_edges = np.histogram(gradient_orientation, bins=8, range=(0, 360))

# Display the edge map and the histogram
plt.figure(figsize=(12, 6))
plt.subplot(121)
plt.imshow(edges, cmap='gray')
plt.title('Edge Map')

plt.subplot(122)
plt.bar(bin_edges[:-1], hist, width=45, align='center')
plt.title('Histogram of Edge Directions')
plt.xlabel('Edge Direction (degrees)')
plt.ylabel('Frequency')
plt.xticks(range(0, 360, 45))
plt.show()

# Print the HED feature vector (histogram)
print("Histogram of Edge Directions (HED) Feature Vector:")
print(hist)
```

5. The Histogram of Intensity Gradients (HIG)

The Histogram of Intensity Gradients (HIG) is a feature extraction technique used in computer vision and image processing. It captures information about the distribution of intensity gradients in an image, which can be valuable for various tasks such as object recognition, texture analysis, and image classification.

Working Mechanism:

1. **Image Preprocessing:**
   - Convert the input image to grayscale if it is not already.
   - Optionally, apply any necessary preprocessing steps like resizing, smoothing, or contrast enhancement.

2. **Gradient Calculation:**
   - Compute the gradient of the image using techniques like the Sobel operator, Scharr operator, or any other gradient calculation method.
   - The gradient represents how the pixel intensities change across the image and is computed separately for both the horizontal (dx) and vertical (dy) directions.

3. **Gradient Magnitude and Orientation:**
   - Calculate the magnitude and orientation of the gradient at each pixel using the following formulas:
   - Magnitude (G): $G = \text{sqrt}(dx^2 + dy^2)$
   - Orientation (θ): $\theta = \text{atan2}(dy, dx)$

4. **Histogram Construction:**
   - Divide the range of gradient orientations (typically 0 to 360 degrees) into bins.
   - For each pixel, determine its gradient orientation and increment the corresponding bin in the histogram.
   - You can use a fixed number of bins or adaptively select the number of bins based on the image content.

5. **Feature Extraction:**
   - The histogram constructed in the previous step represents the distribution of gradient orientations in the image.
   - Optionally, normalize the histogram to make it scale-invariant.
   - The resulting histogram serves as the feature vector, capturing information about the intensity gradients in the image.

6. **Feature Vector:**
   The computed HIG histogram is a feature vector that can be used for various computer vision tasks, such as image classification or object detection.

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

# Read an image (grayscale)
```

```
image = cv2.imread('image.jpg', cv2.IMREAD_GRAYSCALE)

# Compute the gradient using the Sobel operator
dx = cv2.Sobel(image, cv2.CV_64F, 1, 0, ksize=3)
dy = cv2.Sobel(image, cv2.CV_64F, 0, 1, ksize=3)

# Calculate the magnitude and orientation of the gradient
magnitude = np.sqrt(dx**2 + dy**2)
orientation = np.arctan2(dy, dx) * (180 / np.pi)  # Convert radians to degrees

# Create a histogram of gradient orientations
histogram, bins = np.histogram(orientation, bins=9, range=(0, 180))

# Normalize the histogram (optional)
histogram = histogram / histogram.sum()

# Display the histogram
plt.bar(bins[:-1], histogram, width=20)
plt.title("Histogram of Intensity Gradients (HIG)")
plt.xlabel("Gradient Orientation (degrees)")
plt.ylabel("Frequency")
plt.show()

# You can use the computed histogram as a feature vector for your application.
```

6. Texture energy and contrast histograms

Texture energy and contrast histograms are two features commonly used in texture analysis. They provide information about the variation and contrast of textures within an image.

Texture Energy:

- Texture energy measures the uniformity or smoothness of a texture within an image.
- It is calculated as the sum of squared pixel intensities within a local neighborhood.
- Higher energy values indicate a more complex and less uniform texture.

Texture Contrast:

- Texture contrast quantifies the variation in pixel intensities within a local neighborhood.
- It is computed as the standard deviation of pixel intensities in the neighborhood.
- Higher contrast values indicate a greater variation in texture.

Working Mechanism

Image Preprocessing:

- Convert the input image to grayscale if it is not already.
- Optionally, apply any necessary preprocessing steps like resizing, smoothing, or contrast enhancement.

Texture Analysis:

- Define a local neighborhood or window size to analyze texture locally. Common sizes are 3x3 or 5x5.
- For each pixel in the image, extract the pixel values within the defined neighborhood.

Texture Energy Calculation:

- Compute the texture energy by summing the squared pixel values within the neighborhood:
- Texture Energy (E) = $\sum$(pixel values)^2

Texture Contrast Calculation:

- Calculate the texture contrast as the standard deviation of pixel values within the neighborhood:
- Texture Contrast (C) = $\sqrt{}$(Var(pixel values))

Histogram Construction:

- Create histograms for texture energy and contrast separately.
- Divide the range of values into bins for both histograms.
- Count the number of pixels with energy or contrast values falling into each bin.

Feature Extraction:

- The constructed histograms serve as feature vectors.
- These histograms capture information about the distribution of texture energy and contrast in the image.
- These features can be used for texture classification, segmentation, or other texture-related tasks.

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

# Read an image (grayscale)
image = cv2.imread('image.jpg', cv2.IMREAD_GRAYSCALE)

# Define the neighborhood size (e.g., 3x3 window)
neighborhood_size = 3

# Calculate texture energy and contrast using OpenCV filter2D
energy = cv2.filter2D(image ** 2, -1, np.ones((neighborhood_size, neighborhood_size)))
contrast = cv2.filter2D(image, -1, np.ones((neighborhood_size, neighborhood_size)))

# Compute histograms for texture energy and contrast
energy_histogram, energy_bins = np.histogram(energy, bins=256, range=(0, energy.max()))
contrast_histogram, contrast_bins = np.histogram(contrast, bins=256, range=(0, contrast.max()))

# Normalize histograms (optional)
energy_histogram = energy_histogram / energy_histogram.sum()
contrast_histogram = contrast_histogram / contrast_histogram.sum()

# Display the histograms
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(energy_bins[:-1], energy_histogram, color='b')
plt.title("Texture Energy Histogram")
```

```
plt.xlabel("Energy")
plt.ylabel("Frequency")

plt.subplot(1, 2, 2)
plt.plot(contrast_bins[:-1], contrast_histogram, color='r')
plt.title("Texture Contrast Histogram")
plt.xlabel("Contrast")
plt.ylabel("Frequency")

plt.tight_layout()
plt.show()

        # You can use the computed histograms as feature vectors for texture analysis.
```

Task: Texture Analysis for Material Classification (Give Code and Description)

You are working on a project that involves classifying different materials in images, such as wood, metal, and fabric. The goal is to extract texture features from these materials for classification purposes.

- Describe at least two texture feature extraction techniques you would use in this project. Explain how each technique works and provide examples of the types of textures they are suitable for.
- Discuss the advantages and limitations of the selected texture feature extraction techniques in the context of material classification

Filtering & Convolution

Filtering and convolution are fundamental operations in image processing and computer vision. They are used for various tasks, including noise reduction, edge detection, feature extraction, and image enhancement.

Filtering:

- Filtering in image processing involves applying a filter or kernel to an input image.
- A filter is a small matrix of numbers that is convolved with the image to modify the pixel values.
- Each element of the filter represents a weighted contribution to the new pixel value.
- Filtering operations can enhance or suppress certain image features.

Convolution:

- Convolution is the mathematical operation used to apply a filter to an image.
- It involves sliding the filter over the image, element-wise multiplying the filter and the corresponding image region, and summing the results.
- Convolution is expressed as the dot product between the filter and the image region.
- It is performed for every pixel in the image, resulting in a new image called the output or convolved image.

Box Blur

A simple blurring technique that replaces each pixel with the average of its neighboring pixels within a square kernel.
Useful for noise reduction and smoothing.

```
import cv2
import numpy as np
# Read an image
image = cv2.imread('image.jpg')
# Define a box blur kernel
kernel = np.ones((3, 3), dtype=np.float32) / 9
# Apply convolution
blurred_image = cv2.filter2D(image, -1, kernel)
```

Gaussian Blur

Blurring technique using a Gaussian-shaped kernel.
It provides smoother results compared to box blur and is often used for noise reduction.

```
# Define a Gaussian blur kernel
kernel_size = (5, 5)
sigma = 1.0
kernel = cv2.getGaussianKernel(kernel_size[0], sigma)
gaussian_blur = cv2.filter2D(image, -1, kernel)
```

Edge Detection (e.g., Sobel and Scharr)

Detects edges in the image by emphasizing rapid changes in pixel values.
Sobel and Scharr operators are commonly used for edge detection.

```
# Sobel edge detection
sobel_x = cv2.Sobel(image, cv2.CV_64F, 1, 0, ksize=3)
sobel_y = cv2.Sobel(image, cv2.CV_64F, 0, 1, ksize=3)

# Scharr edge detection
scharr_x = cv2.Scharr(image, cv2.CV_64F, 1, 0)
scharr_y = cv2.Scharr(image, cv2.CV_64F, 0, 1)
```

Embossing

Creates a 3D effect by emphasizing the differences in neighboring pixel values.
It can be used for artistic or stylized image effects.

```
# Define an emboss kernel
kernel = np.array([[-2, -1, 0],
                   [-1, 1, 1],
                   [0, 1, 2]], dtype=np.float32)
embossed_image = cv2.filter2D(image, -1, kernel)
```

Convolution Types:

Convolution operations are fundamental in image processing and computer vision. They involve applying a filter (also known as a kernel) to an input image to modify or enhance certain feature.

Standard Convolution:

- Standard convolution, also known as full convolution, is the most common type.
- It processes each pixel in the input image by centering the kernel over it and performing element-wise multiplication and summation.
- The result is placed in the corresponding pixel of the output image.

```
import cv2
import numpy as np
# Read an image
image = cv2.imread('image.jpg', cv2.IMREAD_GRAYSCALE)
# Define a kernel (3x3 example)
        kernel = np.array([[1, 0, -1],
                           [2, 0, -2],
                           [1, 0, -1]], dtype=np.float32)

# Apply standard convolution
result = cv2.filter2D(image, -1, kernel)
```

Valid Convolution (No Padding)

- Valid convolution, also known as no-padding convolution, only processes pixels where the kernel fully overlaps with the input image.
- It produces an output image with smaller dimensions than the input image

```
import cv2
import numpy as np

# Read an image
image = cv2.imread('image.jpg', cv2.IMREAD_GRAYSCALE)

# Define a kernel (3x3 example)
kernel = np.array([[1, 0, -1],
                   [2, 0, -2],
                   [1, 0, -1]], dtype=np.float32)

# Apply valid convolution (no padding)
result = cv2.filter2D(image, -1, kernel, borderType=cv2.BORDER_CONSTANT)
```

Same Convolution (Zero Padding)

- Same convolution adds zero-padding to the input image to ensure that the output image has the same dimensions as the input.
- It is often used to prevent information loss near the image boundaries.

```
import cv2
import numpy as np
# Read an image
image = cv2.imread('image.jpg', cv2.IMREAD_GRAYSCALE)

# Define a kernel (3x3 example)
kernel = np.array([[1, 0, -1],
                   [2, 0, -2],
                   [1, 0, -1]], dtype=np.float32)

# Apply same convolution (zero padding)
result = cv2.filter2D(image, -1, kernel, borderType=cv2.BORDER_REFLECT)
```

Valid Convolution with Strides

- Valid convolution with strides processes pixels where the kernel fully overlaps with the input image, but it skips some pixels based on the specified stride.
- It produces an output image with smaller dimensions than the input, determined by the stride.

```
import cv2
import numpy as np

# Read an image
image = cv2.imread('image.jpg', cv2.IMREAD_GRAYSCALE)

# Define a kernel (3x3 example)
kernel = np.array([[1, 0, -1],
                   [2, 0, -2],
                   [1, 0, -1]], dtype=np.float32)

# Apply valid convolution with a stride of 2
result = cv2.filter2D(image, -1, kernel, strides=(2, 2), borderType=cv2.BORDER_CONSTANT)
```

Edge Detection & its Common Techniques

Edge detection is a fundamental technique in image processing and computer vision used to identify boundaries within an image. These boundaries are often transitions from one object or region to another, representing important information about the image's structure. Edges can correspond to changes in color, intensity, or texture and play a crucial role in various computer vision applications, such as object detection, image segmentation, and feature extraction.

Edge detection is a critical preprocessing step for further analysis of images, as it reduces the amount of data while highlighting essential features. Several edge detection techniques exist, each with its strengths and weaknesses, making them suitable for different scenarios.

Common Edge Detection Techniques

- **Gradient-Based Edge Detection:**
  - ✓ Gradient-based methods identify edges by calculating the gradient (rate of change) of pixel intensities in the image.
  - ✓ Common gradient-based operators include the Sobel, Prewitt, and Scharr operators.
  - ✓ The gradient direction and magnitude help locate edges, and edges typically occur where the gradient magnitude is high.

- **Canny Edge Detector:**
  - ✓ The Canny edge detector is a multi-stage algorithm known for its accuracy and noise reduction capabilities.
  - ✓ It involves Gaussian smoothing to reduce noise, gradient calculation, non-maximum suppression to thin edges, and edge tracking by hysteresis to detect continuous edges.

- **Laplacian of Gaussian (LoG):**
  - ✓ The LoG operator combines Gaussian smoothing and Laplacian edge detection.
  - ✓ It enhances edges by highlighting zero-crossings in the second derivative of the image.

- **Sobel and Scharr Operators:**
  - ✓ Sobel and Scharr operators are gradient-based edge detectors used for approximating the gradient of the image.
  - ✓ They are simple to implement and can be applied in both horizontal and vertical directions.

- **Prewitt Operator:**
  - ✓ The Prewitt operator is another gradient-based method used to detect edges.
  - ✓ It convolves the image with 3x3 kernels to compute gradients in horizontal and vertical directions.

- **Marr-Hildreth Edge Detector (LoG of Gaussian):**
  - ✓ The Marr-Hildreth edge detector uses the LoG operator after Gaussian smoothing.
  - ✓ It emphasizes zero-crossings to locate edges.

- **Sobel-Feldman Operator:**
  - ✓ The Sobel-Feldman operator is a variation of the Sobel operator, emphasizing diagonal edges in addition to horizontal and vertical edges.

- **Edge Detection with Convolutional Neural Networks (CNNs):**
  - ✓ Modern approaches use deep learning techniques, particularly CNNs, for edge detection.
  - ✓ CNN-based models can learn complex edge patterns and adapt to various image domains.

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

```
# Read an image in grayscale
image = cv2.imread('image.jpg', cv2.IMREAD_GRAYSCALE)
# Apply Sobel operator to calculate gradients
gradient_x = cv2.Sobel(image, cv2.CV_64F, 1, 0, ksize=3)  # Gradient in the x-direction
gradient_y = cv2.Sobel(image, cv2.CV_64F, 0, 1, ksize=3)  # Gradient in the y-direction
# Calculate the gradient magnitude and direction
gradient_magnitude = np.sqrt(gradient_x**2 + gradient_y**2)
gradient_direction = np.arctan2(gradient_y, gradient_x)
# Optionally, you can apply a threshold to identify edges
threshold = 100
edges = (gradient_magnitude > threshold).astype(np.uint8) * 255
# Display the original image and detected edges
cv2.imshow('Original Image', image)
cv2.imshow('Detected Edges', edges)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

The Canny Edge Detector

The Canny Edge Detector is a popular and widely used edge detection technique in image processing and computer vision. Developed by John F. Canny in 1986, this technique is known for its accuracy in detecting edges while suppressing noise. It involves a multi-stage process that aims to find the edges in an image.

Key Steps in the Canny Edge Detection Algorithm

Gaussian Smoothing:

- ✓ The first step is to apply Gaussian smoothing to the input image. This step reduces noise in the image, making it less sensitive to small variations in pixel intensity.
- ✓ A Gaussian kernel is convolved with the image, effectively blurring it.

Gradient Calculation:

- ✓ The next step calculates the gradient of the smoothed image. It finds the rate of change of pixel intensities in both the horizontal (x) and vertical (y) directions.
- ✓ Two 3x3 convolution kernels (Sobel operators) are applied to compute the gradient in the x and y directions.
- ✓ The gradient magnitude and direction are then calculated for each pixel.

Non-Maximum Suppression:

- ✓ In this step, the algorithm looks for local maxima in the gradient magnitude.
- ✓ For each pixel, it checks if the gradient magnitude is the maximum among its neighbors in the direction of the gradient.
- ✓ This process thins the edges, retaining only the most prominent ones.

Edge Tracking by Hysteresis:

- ✓ The final step aims to link the remaining edge segments into continuous lines or curves.
- ✓ Two threshold values, a high threshold (T_high) and a low threshold (T_low), are used.
- ✓ Pixels with gradient magnitudes above T_high are considered strong edge points and are definitely part of the edges.

- ✓ Pixels with gradient magnitudes between T_low and T_high are considered weak edge points.
- ✓ Weak edge points are connected to strong edge points if they are adjacent and form continuous edges.
- ✓ The result is a binary edge map with strong and weak edges.

Parameters in the Canny Edge Detector:

- ✓ **Gaussian Kernel Size (σ):** The size of the Gaussian filter kernel used for smoothing. A larger σ results in a smoother image but may lead to loss of fine details.

- ✓ **High Threshold (T_high):** The threshold above which a pixel is considered a strong edge point.

- ✓ **Low Threshold (T_low):** The threshold below which a pixel is rejected as a non-edge point.

The choice of threshold values is crucial and affects the trade-off between edge detection sensitivity and noise suppression. It often requires experimentation and domain knowledge.

```
import cv2
# Read an image in grayscale
image = cv2.imread('image.jpg', cv2.IMREAD_GRAYSCALE)

# Apply Canny edge detection
edges = cv2.Canny(image, threshold1=100, threshold2=200)  # Adjust threshold values as needed

# Display the edge map
cv2.imshow('Canny Edges', edges)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Laplacian of Gaussian (LoG) Edge Detection

The Laplacian of Gaussian (LoG) is an edge detection technique that combines Gaussian smoothing and the Laplacian operator to highlight edges in an image. It is also known as the Marr-Hildreth edge detector and is particularly useful for detecting edges with fine details and reducing noise. Here's how the LoG edge detection process works:

Working Mechanism:

Gaussian Smoothing:

- ✓ The first step is to apply Gaussian smoothing to the input image. This step helps reduce noise and provides a more stable foundation for edge detection.
- ✓ Gaussian smoothing involves convolving the image with a Gaussian kernel, which is a 2D bell-shaped function.

Laplacian Operator:

- ✓ After Gaussian smoothing, the Laplacian operator is applied to the smoothed image. The Laplacian is a second-order derivative operator that calculates the change in intensity within the image.
- ✓ It enhances regions with rapid intensity changes, which typically correspond to edges.

Edge Enhancement:

- ✓ The Laplacian operator highlights edges by emphasizing areas where the intensity changes significantly. Positive values represent bright-to-dark transitions, and negative values represent dark-to-bright transitions.

Zero Crossing Detection:

- ✓ To obtain a binary edge map, zero-crossings in the Laplacian image are detected. Zero-crossings occur where the Laplacian changes sign, indicating the presence of an edge.
- ✓ Edges are localized at the points where the Laplacian changes from positive to negative or vice versa.

```
import cv2
import numpy as np

# Load the image in grayscale
image = cv2.imread('image.jpg', cv2.IMREAD_GRAYSCALE)

# Apply Gaussian smoothing
image_smoothed = cv2.GaussianBlur(image, (5, 5), 0)

# Apply the Laplacian operator
laplacian = cv2.Laplacian(image_smoothed, cv2.CV_64F)

# Find zero crossings
edges = cv2.Canny(laplacian, threshold1=30, threshold2=100)

# Display the results
cv2.imshow('Original Image', image)
cv2.imshow('LoG Edge Detection', edges)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Zero Crossing Edge Detection

- ✓ Zero crossing methods identify edges by locating points where the intensity changes sign (e.g., from positive to negative or vice versa) in the image.
- ✓ They are sensitive to noise and require post-processing to refine edge maps

The Marr-Hildreth Edge Detector

The Marr-Hildreth Edge Detector, also known as the LoG (Laplacian of Gaussian) edge detector, is an edge detection technique that combines Gaussian smoothing and Laplacian edge detection. It is designed to locate edges in an image by emphasizing zero-crossings in the second derivative of the image.

Marr-Hildreth Edge Detector (LoG of Gaussian) works:

Gaussian Smoothing:

- ✓ The first step is to apply Gaussian smoothing to the input image. This step reduces noise in the image and helps in highlighting important features.
- ✓ The Gaussian kernel is convolved with the image to create a smoothed version of the original image.

Laplacian Operator:

- ✓ After Gaussian smoothing, the Laplacian operator (also known as the second derivative operator) is applied to the smoothed image.
- ✓ The Laplacian operator computes the Laplacian of the image, which represents the rate of change of pixel intensities.
- ✓ Mathematically, the Laplacian operator is defined as the sum of the second derivatives in both the x and y directions.

Zero-Crossings Detection:

- ✓ The key feature of the Marr-Hildreth Edge Detector is its emphasis on zero-crossings in the Laplacian image.
- ✓ Zero-crossings occur at points where the intensity changes sign (from positive to negative or vice versa).
- ✓ These zero-crossings are indicative of potential edge locations in the original image.

Thresholding:

- ✓ To obtain a binary edge map, a thresholding step is often applied to the zero-crossings image.
- ✓ Pixels with values near zero (close to a zero-crossing) are considered as edges, while other pixels are suppressed.

```
import cv2
import numpy as np

# Read the image
image = cv2.imread('image.jpg', cv2.IMREAD_GRAYSCALE)

# Apply Gaussian smoothing
smoothed_image = cv2.GaussianBlur(image, (5, 5), 0)

# Apply Laplacian operator for edge detection
laplacian = cv2.Laplacian(smoothed_image, cv2.CV_64F)

# Find zero-crossings in the Laplacian image
edges = cv2.threshold(np.abs(laplacian), 30, 255, cv2.THRESH_BINARY)[1]

# Display the original image and edge map
cv2.imshow('Original Image', image)
cv2.imshow('Edge Map (Marr-Hildreth)', edges)

# Wait for a key press and then close the windows
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Corner Detection & its Common Techniques

Corner Detection:
Corner detection is a fundamental technique in computer vision used to identify significant points or corners in an image. Corners represent locations where the intensity of an image changes in multiple directions, making them suitable for various applications like image registration, object recognition, and feature tracking.

The primary objective of corner detection is to locate points where the local gradient of intensity exhibits significant changes in multiple directions. Such points are invariant under translation, rotation, and scale changes, making them robust features for image analysis.

Common Techniques for Corner Detection:

1. **Harris Corner Detector:**
   - ✓ The Harris corner detector is one of the earliest and most widely used methods for corner detection.
   - ✓ It calculates a "cornerness" score for each pixel by considering the variation of intensity in all directions within a local neighborhood.
   - ✓ High corner scores indicate the presence of a corner.
2. **Shi-Tomasi Corner Detector:**
   - ✓ The Shi-Tomasi corner detector is an improvement over the Harris detector, using a slightly different scoring function.
   - ✓ It selects corners based on a threshold applied to the minimum eigenvalue of a 2x2 gradient matrix.
3. **FAST (Features from Accelerated Segment Test):**
   - ✓ FAST is a machine-learning-based corner detector designed for real-time applications.
   - ✓ It uses a decision tree to classify pixels as corners or non-corners based on the intensity values of surrounding pixels.
4. **SIFT (Scale-Invariant Feature Transform):**
   - ✓ SIFT detects scale-invariant keypoints, which often correspond to corners.
   - ✓ It identifies keypoints at multiple scales and orientations, making it robust to variations in scale and rotation.
5. **SURF (Speeded-Up Robust Features):**
   - ✓ SURF is another method for detecting scale and rotation-invariant keypoints, including corners.
   - ✓ It uses integral images to speed up computation and performs similarly to SIFT but with faster processing.
6. **ORB (Oriented FAST and Rotated BRIEF):**
   - ✓ ORB combines the FAST keypoint detector with the BRIEF (Binary Robust Independent Elementary Features) descriptor.
   - ✓ It's designed for real-time applications and provides a balance between speed and accuracy.

```
import cv2
import numpy as np

# Read an image
image = cv2.imread('image.jpg', cv2.IMREAD_GRAYSCALE)

# Define parameters
block_size = 2  # Neighborhood size for computing the structure tensor
```

```
k = 0.04  # Empirical constant (usually in the range [0.04, 0.06])

# Compute Harris corner scores
corner_scores = cv2.cornerHarris(image, block_size, 3, k)  # Add '3' as the third argument
# Threshold the corner scores to identify corners
threshold = 0.01 * corner_scores.max()
corners = np.where(corner_scores > threshold)

# Draw circles at the detected corners
```

```
#Detect Shi-Tomasi corners
corners = cv2.goodFeaturesToTrack(image, maxCorners=100, qualityLevel=0.01, minDistance=10)
```

```
# Create a FAST detector object with specified parameters
fast = cv2.FastFeatureDetector_create(threshold=40, nonmaxSuppression=True)
# Detect FAST keypoints
keypoints = fast.detect(image, None)
```

```
# Initialize the SIFT detector
sift = cv2.SIFT_create()

# Detect keypoints and compute descriptors
keypoints, descriptors = sift.detectAndCompute(image, None)
```

```
# Create a SURF object
surf = cv2.SURF_create()

# Detect keypoints and compute descriptors
keypoints, descriptors = surf.detectAndCompute(image, None)
```

ORB (Oriented FAST and Rotated BRIEF) is a keypoint detector and descriptor combination that is particularly designed for real-time computer vision applications. It was developed to provide an efficient and robust alternative to traditional feature detection and matching techniques like SIFT and SURF. ORB has gained popularity due to its computational efficiency and effectiveness in various computer vision tasks.

Working Mechanism:

Feature Detection (FAST): ORB starts with the FAST (Features from Accelerated Segment Test) detector, which identifies potential keypoints (interest points) in an image. FAST is known for its computational speed and operates by comparing the intensity of a central pixel to the intensities of pixels in a circle around it.

Feature Description (BRIEF): Once keypoints are detected, ORB computes a feature descriptor for each keypoint using the BRIEF (Binary Robust Independent Elementary Features) descriptor. BRIEF represents the local neighborhood around a keypoint as a binary string, which makes it efficient to compute and match.

Rotation and Scale Invariance: ORB addresses rotation invariance by assigning an orientation to each keypoint. This orientation is determined based on the local intensity pattern around the keypoint. ORB is also scale-invariant, allowing it to work with features at different scales.

Matching: To find correspondences between keypoints in different images, ORB uses techniques like the Hamming distance to compare binary descriptors efficiently. It can also employ techniques like the RANSAC algorithm to remove outliers and estimate geometric transformations.

```
import cv2
import numpy as np

# Load two images
image1 = cv2.imread('dog.jpeg', cv2.IMREAD_GRAYSCALE)
image2 = cv2.imread('dogg.jpg', cv2.IMREAD_GRAYSCALE)

# Create ORB detector
orb = cv2.ORB_create()

# Find keypoints and descriptors in both images
keypoints1, descriptors1 = orb.detectAndCompute(image1, None)
keypoints2, descriptors2 = orb.detectAndCompute(image2, None)

# Create a Brute Force Matcher
bf = cv2.BFMatcher(cv2.NORM_HAMMING, crossCheck=True)

# Match descriptors
matches = bf.match(descriptors1, descriptors2)

# Sort them in ascending order of distance
matches = sorted(matches, key=lambda x: x.distance)

# Draw the first 10 matches
match_image = cv2.drawMatches(image1, keypoints1, image2, keypoints2, matches[:10], outImg=None)

# Display the matches
cv2.imshow('Matches', match_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Lab Task 2: Shi-Tomasi Corner Detection

In another project, you need to find the most prominent corners in an image for a feature matching task. You choose the Shi-Tomasi Corner Detection method for this purpose. Perform the following tasks:

- ✓ Load a different image (choose a distinct image from the previous task).
- ✓ Implement the Shi-Tomasi Corner Detection algorithm to find the most prominent corners in the image.
- ✓ Display the original image with the detected corners marked as green points.
- ✓ Adjust the number of corners to be detected by changing the quality level parameter and observe the results.

Lab Task:

1. **Medical Image Analysis for Tumor Detection (Code & Description)**

You are part of a medical research team working on a project to detect tumors in medical images, specifically mammograms. Feature extraction plays a critical role in identifying potential tumor regions.

- ✓ Explain how the concept of edge detection can be applied as a feature extraction technique for tumor detection in mammograms. Provide a step-by-step description of how edge detection can be used in this context.
- ✓ In addition to edge detection, propose one more feature extraction technique that can enhance tumor detection in medical images. Describe this technique, its working mechanism, and why it is suitable for this task.

These questions focus on feature extraction techniques and their application in different domains, encouraging the understanding of how to choose and apply appropriate techniques for specific image analysis tasks.

2. **Lab Task 1: Harris Corner Detection**

   You are working on an image processing project where you need to detect corners in an image. You decide to use the Harris Corner Detection method. Perform the following tasks:

   - ✓ Load an image (you can use any image you prefer).
   - ✓ Implement the Harris Corner Detection algorithm to detect corners in the image.
   - ✓ Display the original image with the detected corners marked as red points.
   - ✓ Experiment with different threshold values and observe how they affect the corner detection results.

3. **Lab Task 3: Corner Detection in Real-time Video**

   You are working on a real-time computer vision application where you need to detect corners in a live video stream. Implement corner detection in real-time using the Harris or Shi-Tomasi method. Perform the following tasks:

   - ✓ Access the video feed from your computer's webcam or use a pre-recorded video.
   - ✓ Implement either the Harris or Shi-Tomasi Corner Detection algorithm to process the video frames and detect corners.
   - ✓ Display the live video stream with the detected corners marked as points in real-time.
   - ✓ Experiment with different parameters to optimize corner detection for real-time processing.

4. **Lab Task 4: Corner Detection for Image Stitching**

   You are developing an image stitching application that combines multiple images into a panorama. Corner detection is a crucial step for finding matching features between images. Perform the following tasks:

   - ✓ Choose multiple images that you want to stitch together to create a panorama.
   - ✓ Implement either the Harris or Shi-Tomasi Corner Detection algorithm to detect corners in each image.
   - ✓ Display each image with the detected corners marked as points.
   - ✓ Use the detected corners as key features for image matching and stitching.
   - ✓

5. **Lab Task: Feature Detection and Matching using ORB Detector**

You are working on a computer vision project that involves detecting and matching features between two images. You decide to use the ORB (Oriented FAST and Rotated BRIEF) detector and descriptor for this task.

Task Instructions:

- ✓ Load Images:
- ✓ Select two different images (e.g., natural scenes, objects, or patterns) that have some common features but may have variations like rotation, scaling, or perspective changes.
- ✓ Load both images using OpenCV.

Feature Detection with ORB:

- ✓ Implement the ORB feature detector and descriptor using OpenCV.
- ✓ Detect key points and compute descriptors for both images using ORB.
- ✓ Matching Features:
- ✓ Implement feature matching using the Brute-Force Matcher or FLANN (Fast Library for Approximate Nearest Neighbors) Matcher.

Match the descriptors of key points between the two images.

- ✓ Draw Matches:
- ✓ Draw the matches between key points on a new image to visualize the feature matches.
- ✓ You can use lines or circles to connect matched points.
- ✓ Display Results:

Display the following:

- ✓ The two original images side by side.
- ✓ The image with key points marked (you can use circles to represent key points).
- ✓ The image with matched key points marked (showing the connections between matching points).
- ✓ Experiment and Analysis:
- ✓ Experiment with different sets of images with varying transformations (rotation, scaling, perspective changes) to observe how well ORB handles feature matching.
- ✓ Analyze the performance of ORB in terms of robustness and accuracy in matching features.

Expected Output: