סמינר - Image Processing & Computer Vision

ליאור שילון

Seminar Outline

What We Will Cover Today:

This 3-hour workshop takes you from the fundamentals of image processing to advanced deep learning techniques for object detection.
We'll build a strong foundation in computer vision using OpenCV and Python, then progressively advance through machine learning pipelines to state-of-the-art neural network architectures.

Journey Through Computer Vision

Part 1: Foundation

- Setting up Anaconda, OpenCV and essential libraries
- Understanding pixels, coordinates, and image representation
- Basic image manipulation and drawing operations

Part 2: Image Processing Techniques

- Transformations: resize, rotate, flip, crop
- Filtering and noise reduction
- Edge detection and contour analysis
- Morphological operations

Part 3: Machine Learning Pipeline

- Feature extraction methods (HOG, GLCM, LBP)
- Feature selection strategies
- Model training and deployment

ליאור שילון

Journey Through Computer Vision

Not Covered Today.
Recommended for
Self-Study.

Part 4: Deep Learning Foundations

- Artificial Neural Networks and Perceptrons
- Convolutional Neural Networks (CNNs)
- Training models with TensorFlow
- Practical example: Handwritten digit recognition

Part 5: Object Detection

- R-CNN family (R-CNN, Fast R-CNN, Faster R-CNN)
- YOLO architecture and variations
- SSD (Single Shot Multibox Detector)
- Training custom object detectors

Prerequisites:
VSCode
Python

ליאור שילון

Core Libraries

OpenCV

- Install: `pip install opencv-python opencv-contrib-python`

NumPy

- Install: `pip install numpy`
- For array operations

Matplotlib

- Install: `pip install matplotlib`
- For visualization

SciPy

- Install: `pip install scipy`
- For scientific computing

TensorFlow 2.x

- CPU version(CPU and GPU support): `pip install tensorflow`
- M1 Mac: `pip install tensorflow-macos`

ScikitImage

- Install: `pip install scikit-image`

Scikit-learn

- Install: `pip install scikit-learn`

ליאור שילון

Installing Anaconda

Step 1: Go to anaconda website

Step 2: Press the “Get Started” button

Step 3: Sign-up

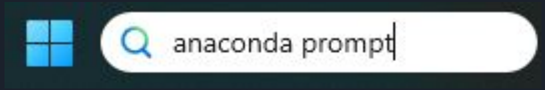
Step 4: Download the distribution installer

Step 5: Install Anaconda

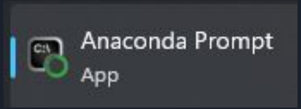
Step 6 (Optional): Add Anaconda to system PATH (check how to do so)

Setting Up Anaconda Virtual Environment

Step 1: Search for “anaconda prompt” in the windows search bar


anaconda prompt

Step 2: Open it


Anaconda Prompt
App

Step 3: Create an environment called cv_seminar_env with python 3.10

```
conda create -n cv_seminar_env python=3.10 -y
```

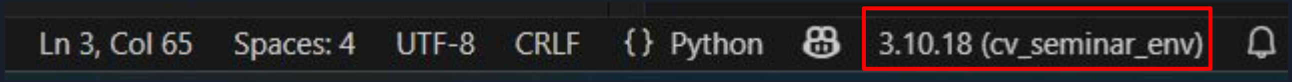
Step 4: Activate the environment

```
conda activate cv_seminar_env
```

Step 5: Install the libraries (copy in one line)

```
pip install opencv-python==4.10.0.84 opencv-contrib-python==4.10.0.84
numpy==1.26.4 matplotlib==3.9.2 scipy==1.13.1 tensorflow==2.17.0
scikit-image==0.24.0 scikit-learn==1.5.2
```

Step 6: Open VSCode and set the interpreter to the virtual env


Ln 3, Col 65 Spaces: 4 UTF-8 CRLF {} Python 3.10.18 (cv_seminar_env)

ליאור שילון

Part 1: Core Concepts of Image Processing

Understanding Pixels, Coordinates, and Basic Operations

ליאור שילון

What is a Digital Image?

Fundamental Definition

A digital image is a 2D array of numbers. Each number represents the intensity or color at a specific location in the image. These individual elements are called pixels (picture elements).

Pixels: The Building Blocks

Every digital image is composed of pixels arranged in a grid. Each pixel contains numerical values that represent:

- Grayscale images: One value (0-255) representing brightness
- Color images: Three values for RGB (Red, Green, Blue) channels
- Images with transparency: Four values (RGBA, where A is alpha/transparency)

What is a Digital Image?

Grayscale Pixel Values

- 0 = Black (no light)
- 255 = White (maximum light)
- 1-254 = Shades of gray

Color Pixel Values (RGB)

Color	R	G	B
Pure Red	255	0	0
Pure Green	0	255	0
Pure Blue	0	0	255
White	255	255	255
Black	0	0	0
Yellow	255	255	0

Image Coordinate System

⚠️ Key Difference from Math:

In image processing, the coordinate system origin (0,0) is at the top-left corner, not bottom-left like in mathematics!

- X-axis: Increases from left to right (columns)
- Y-axis: Increases from top to bottom (rows)

Understanding Pixel Coordinates

Each pixel can be accessed using its (x, y) coordinates:

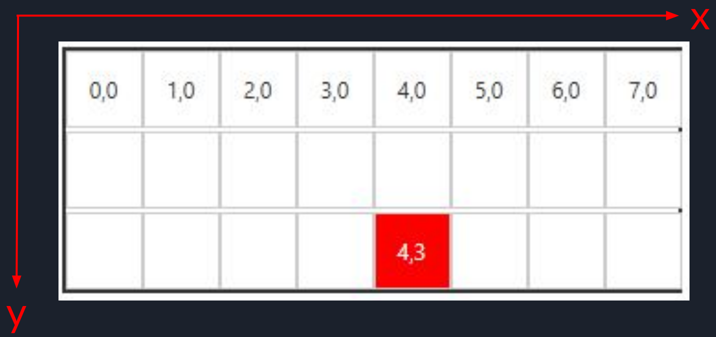
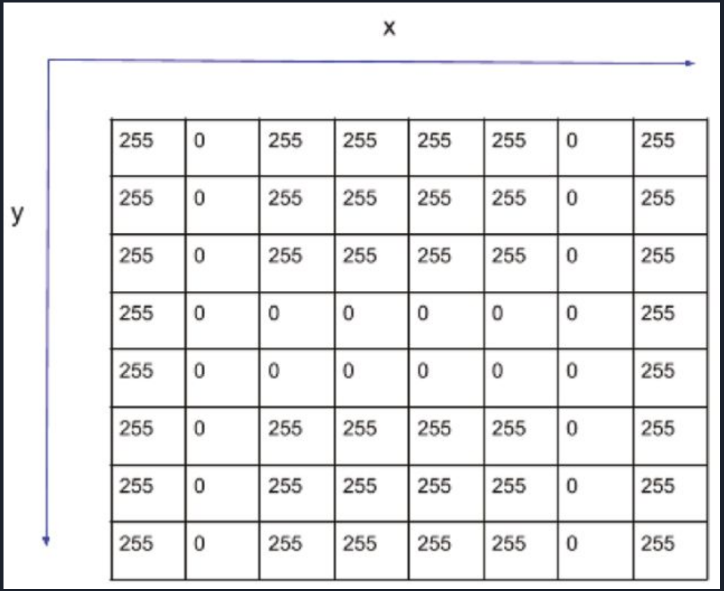


Image Coordinate System

Example: Letter 'H' in 8×8 Grid

Let's visualize how the letter 'H' appears as pixel values:



255	0	255	255	255	255	0	255
255	0	255	255	255	255	0	255
255	0	255	255	255	255	0	255
255	0	0	0	0	0	0	255
255	0	0	0	0	0	0	255
255	0	255	255	255	255	0	255
255	0	255	255	255	255	0	255
255	0	255	255	255	255	0	255

What is the pixel value at position:

- (1,4)?
- (2,2)?

Open:

`Part1_foundation.ipynb -> H letter example`

ליאור שילון

Loading and Exploring Images with OpenCV

Your First OpenCV Program

Open:
`Part1_foundation.ipynb -> Your First OpenCV Program`

Breaking Down the Code

`cv2.imread()` - Loads image from disk into a NumPy array
`image.ndim`- Number of dimensions (3 for color images)
`image.shape`- Returns (height, width, channels)
`image.size`- Total number of pixels × channels
`cv2.imshow()` - Displays image in a window
`cv2.waitKey(0)` - Waits for keyboard input

💡 Important Notes:

- OpenCV loads images in BGR format, not RGB!
- Image shape is (height, width, channels), not (width, height)
- Size = 259 × 415 × 3 = 322,455 total values

ליאור שילון

Accessing and Manipulating Pixels

Reading and Writing Pixel Values

Open:
`Part1_foundation.ipynb -> Access and Manipulate Image Pixels`

More Efficient Pixel Manipulation

```
# Better way - using array slicing
image[0:100, 0:100] = (255, 255, 0)   # Square

# Set a rectangular region to red
image[50:150, 200:400] = (0, 0, 255)  # Red in BGR

# Create a horizontal line (single row of pixels)
image[200, :] = (0, 255, 0)  # Green line

# Create a vertical line (single column)
image[:, 320] = (255, 0, 0)  # Blue line
```

ליאור שילון

Drawing Shapes on Images

OpenCV provides convenient functions to draw geometric shapes:

Drawing Lines

Open:

```
Part1_foundation.ipynb -> Drawing Shapes -> Drawing Lines
```

OpenCV’s line() function:

```
cv2.line(image, start, end, color, thickness)
```

which takes the following arguments:
– Image NumPy array. This is the image on which we are drawing the line.
– The start coordinates of the line in (x, y) format
– The end coordinates of the line in (x, y) format
– The color tuple in BGR format.
– The thickness of the line. (This is optional. If you do not pass this argument, the line will have a default thickness of 1.)

Drawing Shapes on Images

OpenCV provides convenient functions to draw geometric shapes:

Drawing Rectangles

Open:
`Part1_foundation.ipynb -> Drawing Shapes -> Drawing Rectangles`

OpenCV’s rectangle() function:

```
cv2.rectangle(image, start, end, color, thickness)
```

which takes the following arguments:
– NumPy array that holds the pixel values of the image
– The start coordinates (top-left corner of the rectangle) in (x, y) format
– The end coordinates (bottom-right of the rectangle) in (x, y) format
– The color of the outline in BGR format
– The thickness of the outline

ליאור שילון

Drawing Shapes on Images

OpenCV provides convenient functions to draw geometric shapes:

Drawing Circles

Open:
`Part1_foundation.ipynb -> Drawing Shapes -> Drawing Circles`

OpenCV’s circle() function:

```
cv2.circle(canvas, center, radius, color, thickness)
```

which takes the following arguments:
– The image on which to draw the circle. This is our NumPy array containing the image pixels.
– The coordinates of the center of the circle in (x, y) format.
– The radius of the circle.
– The color of the outline of the circle in BGR.
– The thickness of the outline.

ליאור שילון

Key Takeaways from Part 1

Image Fundamentals:

- Images are 2D arrays of pixel values
- Grayscale: 1 value per pixel (0-255)
- Color: 3 values per pixel (BGR in OpenCV)

Coordinate System:

- Origin (0,0) is at top-left
- X increases rightward, Y increases downward
- Access pixels: `image[y, x]` or `image[row, column]`

OpenCV Operations:

- `cv2.imread()` - Load images
- `cv2.imshow()` - Display images
- `cv2.waitKey()` - Wait for input
- `cv2.imwrite()` - Save images

Drawing Functions:

- `cv2.line()` - Draw lines
- `cv2.rectangle()` - Draw rectangles
- `cv2.circle()` - Draw circles

❌ Forgetting that OpenCV uses BGR, not RGB
❌ Confusing (x,y) coordinates with [row, column] indexing
❌ Not calling cv2.waitKey() after imshow()
❌ Forgetting to destroy windows with cv2.destroyAllWindows()
❌ Using thickness=0 (use -1 for filled, positive for outline)

ליאור שילון

Key Takeaways from Part 1

Image Fundamentals:

- Images are 2D arrays of pixel values
- Grayscale: 1 value per pixel (0-255)
- Color: 3 values per pixel (BGR in OpenCV)

Coordinate System:

- Origin (0,0) is at top-left
- X increases rightward, Y increases downward
- Access pixels: `image[y, x]` or `image[row, column]`

OpenCV Operations:

- `cv2.imread()` - Load images
- `cv2.imshow()` - Display images
- `cv2.waitKey()` - Wait for input
- `cv2.imwrite()` - Save images

Drawing Functions:

- `cv2.line()` - Draw lines
- `cv2.rectangle()` - Draw rectangles
- `cv2.circle()` - Draw circles

Part 2: Image Processing Techniques

Transformations, Filtering, Edge Detection & Morphological Operations

ליאור שילון

Image Transformations

Resizing

Let's start with our first transformation, resizing.
To resize an image, we increase or decrease the height and width of the image.
An important concept to remember when resizing an image is aspect ratio, which is the proportion of width to height and is calculated by dividing width by height.
The formula for calculating the aspect ratio is as follows:

```
aspect ratio = width/height
```

A square image has an aspect ratio of 1:1, and an aspect ratio of 3:1 means the width is three times larger than the height.
If an image's height is 300px and the width is 600px, its aspect ratio is 2:1.
We will see:

- Resize an image to a desired size in pixels while maintaining the aspect ratio. In other words, if you know the desired height of the image, you can compute the corresponding width using the aspect ratio from the original size of the image.
- Resize an image by a factor. For example, enlarge the image width by a factor of 1.5 or the height by a factor of 2.5.

Open:

```
Part2_image_processing_techniques.ipynb -> Image Transformations -> Resizing
```

Image Transformations

Translation

Translation moves an image along the x and y axes. Think of it as sliding your image left/right or up/down on a canvas.

Why Use Translation

- Aligning multiple images for comparison
- Creating panoramas by shifting overlapping images
- Data augmentation for machine learning
- Adjusting object positions in scenes

How It Works

Translation uses a 2x3 transformation matrix that defines the movement:

Translation Matrix = | 1 0 tx |
| 0 1 ty |

Where:
- tx = horizontal shift (positive = right, negative = left)
- ty = vertical shift (positive = down, negative = up)

Open:

`Part2_image_processing_techniques.ipynb -> Image Transformations -> Translation`

ליאור שילון

Image Transformations

Rotation

Rotation spins an image around a specified center point by a given angle.
The image can rotate clockwise (negative angle) or counter-clockwise (positive angle).

The Mathematics Behind Rotation

OpenCV uses a rotation matrix that considers both the angle and the center point:

Rotation Matrix = | cos(θ), -sin(θ) , cx(1-cos(θ)) + cy·sin(θ) |
| sin(θ), cos(θ) , cy(1-cos(θ)) - cx·sin(θ) |

Where:
- θ = rotation angle
- (cx, cy) = center of rotation

Open: `Part2_image_processing_techniques.ipynb -> Image Transformations -> Rotation`

OpenCV's warpAffine() function:

```
cv2.warpAffine(image, rotationMatrix, (image.shape[1], image.shape[0]))
```

The NumPy representation of the image we intend to move.
- The translation matrix that defines the movement direction and the amount of the movement.
- The last argument is a tuple that has the width and height of the canvas within which we want to move our image. In this example, we are keeping the canvas size the same as the original height and width of the image.

ליאור שילון

Image Transformations

Flipping

Flipping creates a mirror image along the horizontal or vertical axis.
It's one of the simplest yet most useful transformations!

Flip Codes in OpenCV

Flip Code	Direction	Description
0	Vertical	Flip around x-axis (upside down)
1 (or any positive)	Horizontal	Flip around y-axis (left-right mirror)
-1 (or any negative)	Both	Flip both horizontally and vertically

Open: `Part2_image_processing_techniques.ipynb -> Image Transformations -> Flipping`

ליאור שילון

Image Transformations

Cropping

Cropping removes unwanted outer areas of an image.
Since OpenCV images are NumPy arrays, we use array slicing!

Understanding Array Slicing for Cropping

```
# Remember: images are stored as img[row, column] or img[y, x]
# Slicing syntax: img[y_start:y_end, x_start:x_end]
cropped = img[y1:y2, x1:x2]
# Where:
# y1, y2 = top and bottom boundaries (rows)
# x1, x2 = left and right boundaries (columns)
```

Open: `Part2_image_processing_techniques.ipynb -> Image Transformations -> Cropping`

⚠️ **Common Mistake:**
Remember that OpenCV uses (x, y) for points but arrays use [y, x] for indexing!

ליאור שילון

Image Arithmetic & Bitwise Operations

Image Arithmetic Operations

Image arithmetic involves mathematical operations (addition, subtraction) on pixel values.
But there's a catch - pixel values must stay within 0-255!
Note that to add two images, they must be of the same depth and type

Addition

Operation Type	Example: 230 + 30	Result	Used By
Saturated (Clamping)	min(230 + 30, 255)	255	OpenCV
Modulo (Wrapping)	(230 + 30) % 256	4	NumPy

Open: `Part2_image_processing_techniques.ipynb -> Image Arithmetic & Bitwise Operations -> Addition`

Image Arithmetic & Bitwise Operations

Subtraction

When we subtract two images, it is important to note that the two images must be of the same size and depth.

- Subtracting an image from itself → all pixel values = 0 → black image
- Used to detect changes - if images are identical, result is black
- Also helps to remove shadows or uneven lighting
- Useful for change detection and background correction

Open: `Part2_image_processing_techniques.ipynb -> Image Arithmetic & Bitwise Operations -> Subtraction`

💡 **Pro Tip**:
Image subtraction is perfect for motion detection!
Subtract consecutive frames from a video to see what moved.

ליאור שילון

Image Arithmetic & Bitwise Operations

Bitwise Operations

Bitwise operations work on individual bits of pixel values.
They're essential for masking - selectively showing or hiding parts of an image!

The Four Essential Bitwise Operations

Operation	Symbol	Result	Use Case
AND	&	1 only if both are 1	Masking regions
OR	|	1 only if both are 1	Combining regions
XOR	^	1 if different	Finding differences
NOT	~	Inverts bits	Inverting masks

Open: `Part2_image_processing_techniques.ipynb -> Image Arithmetic & Bitwise Operations -> Bitwise Operations`

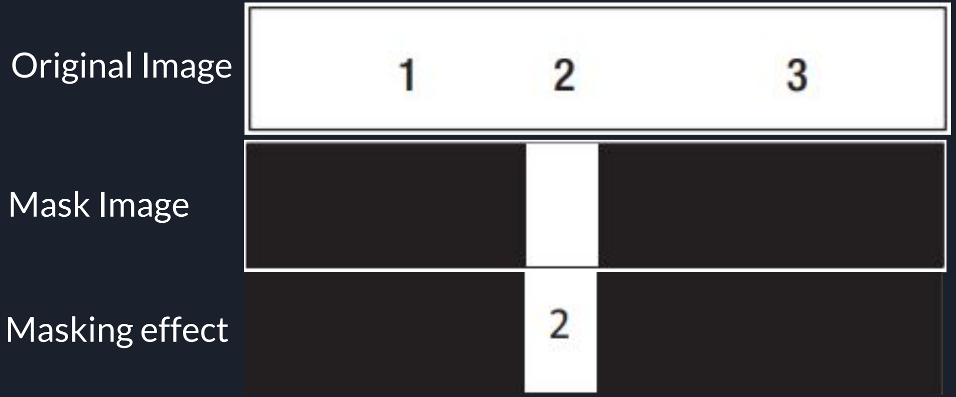
ליאור שילון

Image Arithmetic & Bitwise Operations

Masking

Masking is one of the most powerful techniques in computer vision.
Masking refers to the “hiding” or “filtering” of an image.
When we mask an image, we hide a portion of the image with some other image.
In other words, we put our focus on a portion of the image by applying a mask on the remaining portion of the image.

Try to think about the algorithm



Open: `Part2_image_processing_techniques.ipynb -> Image Arithmetic & Bitwise Operations -> Masking`

ליאור שילון

Image Filtering & Blurring

Image Blurring Techniques

Blurring smooths an image by averaging pixel values with their neighbors.
It's like looking through frosted glass - details become softer!

Why Blur Images

- Noise Reduction: Remove random pixel variations
- Privacy: Hide sensitive information (faces, text)
- Artistic Effects: Create depth of field, motion blur
- Preprocessing: Prepare images for edge detection

Image Filtering & Blurring

Mean Filtering or Averaging

- A small k×k window (called a kernel) moves over the image
- Each center pixel is replaced by the average of neighboring pixels
- Kernel size is usually odd (e.g., 3×3, 5×5) to have a clear center
- Larger kernels → more blur

Open: `Part2_image_processing_techniques.ipynb -> Image Filtering & Blurring -> Mean Filtering/Averaging`

Image Filtering & Blurring

Gaussian Filtering

- Used to reduce Gaussian noise (Intensity variation that follows a Gaussian normal distribution) and create natural blurring
- Uses a Gaussian kernel instead of a simple average kernel
- Kernel defined by height, width, and standard deviations (σ_X, σ_Y)
- Function: `cv2.GaussianBlur(image, (k, k), sigmaX, sigmaY)`
  - If sigmaY not given → equals sigmaX
  - If both = 0 → values auto-calculated from kernel size
  - Produces smoother, more realistic blur than averaging

$$G_0(x,y) = Ae^{\frac{-(x-\mu_x)^2}{2\sigma_x^2}+\frac{-(y-\mu_y)^2}{2\sigma_y^2}}$$

Open: `Part2_image_processing_techniques.ipynb -> Image Filtering & Blurring -> Gaussian Filtering`

ליאור שילון

Image Filtering & Blurring

Median Blurring

- Effective for removing salt-and-pepper noise (Random occurrences of black and white pixels)
- Similar to averaging, but replaces the center pixel with the median of neighbors
- Function: `cv2.medianBlur(image, k)`
- k = kernel size (odd number like 3, 5, 7)
- Preserves edges better than mean blurring

Open: `Part2_image_processing_techniques.ipynb -> Image Filtering & Blurring -> Median Blurring`

ליאור שילון

Image Filtering & Blurring

Bilateral Blurring

Blurs while preserving edges (unlike other blurs)
Uses two Gaussian functions:

- The first Gaussian function considers the spatial neighbors (pixels in x and y space that are close together)
- The second Gaussian function considers the pixel intensity of the neighboring pixels

This makes sure that only those pixels that are of similar intensity to the central pixel are considered for blurring, leaving the edges intact as the edges tend to have higher intensity compared to other pixels.

Function:

- `cv2.bilateralFilter(image, diameter, color, space)`

Slower, but produces superior, edge-preserving results

Open: `Part2_image_processing_techniques.ipynb -> Image Filtering & Blurring -> Bilateral Blurring`

ליאור שילון

Binarization & Thresholding

Image Thresholding Techniques

Thresholding converts a grayscale image to binary (black and white) by comparing each pixel to a threshold value. Pixels above become white (255), below become black (0).

Simple Thresholding

Manually choose a threshold value (T)

- Pixels > T → 255 (white)
- Pixels ≤ T → 0 (black)

Inverse mode: reverses the above (white ↔ black)

Function:

```
cv2.threshold(image, T, maxValue, method)
```

- Methods:
  - THRESH_BINARY
  - THRESH_BINARY_INV

Returns: (threshold value, binary image)

If we are processing a large number of images and want to adjust the threshold values based on the image type and intensity variations, simple thresholding may not be the ideal method

Open: `Part2_image_processing_techniques.ipynb -> Binarization & Thresholding -> Simple Thresholding`

ליאור שילון

Binarization & Thresholding

Adaptive Thresholding

- Automatically adjusts the threshold for each pixel
- Threshold value depends on the local neighborhood
- Handles uneven lighting and intensity variations effectively
- Different image regions can have different thresholds
- Function:
  - `cv2.adaptiveThreshold(image, maxValue, method, type, blockSize, C)`
  - `image → The grayscale image to be binarized`
  - `maxValue → The value assigned to pixels above the threshold (usually 255)`
  - `method → How the threshold is computed for each neighborhood (e.g., mean or Gaussian)`
  - `type → Binarization type (THRESH_BINARY or THRESH_BINARY_INV)`
  - `blockSize → Size of the local neighborhood used to calculate the threshold`
  - `C → Constant subtracted from the local threshold to fine-tune results`

Open: `Part2_image_processing_techniques.ipynb -> Binarization & Thresholding -> Adaptive Thresholding`

ליאור שילון

Binarization & Thresholding

Otsu's Binarization

- Automatically finds the best global threshold using the image histogram
- Maximizes separation between foreground and background pixels
- Use in OpenCV:

```
T , binary = cv2.threshold(image, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU)
```

- No need to manually choose a threshold value

Open: `Part2_image_processing_techniques.ipynb -> Binarization & Thresholding -> Otsu's Binarization`

⚠️ **When to Use Each Method**:
Simple: Uniform lighting, known threshold
Adaptive: Varying lighting, shadows
Otsu: Bimodal histogram, automatic threshold needed

ליאור שילון

Edge Detection & Contours

Edge Detection Techniques

Edge detection identifies points where image brightness changes sharply.
These usually correspond to object boundaries, making them crucial for object recognition!

Gradient-Based Edge Detection

- Detects edges by measuring intensity changes (gradients)
- Highlights regions with rapid pixel value changes
- Two main methods in OpenCV:
  - Sobel Derivatives
  - Laplacian Derivatives

Sobel Derivatives

- Combines Gaussian smoothing + Sobel differentiation
- More noise-resistant due to smoothing
- Detects edges in horizontal (xorder=1, yorder=0) or vertical (xorder=0, yorder=1) directions
- ksize defines kernel size (e.g., 3, 5, 7)
- If ksize = -1, OpenCV applies a 3×3 Scharr filter — gives sharper results

Open: `Part2_image_processing_techniques.ipynb -> Edge Detection & Contours -> Gradient-Based Edge Detection`

ליאור שילון

Edge Detection & Contours

Laplacian Derivatives

- Computes the second derivative of pixel intensities
- Detects edges where intensity changes rapidly in any direction
- OpenCV function: cv2.Laplacian()
- Key Parameters:
  - image → Input image for edge detection
  - ddepth → Data type for output (usually cv2.CV_64F to support negative and floating-point values)

$$\text{Laplace}(f) = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

Open: `Part2_image_processing_techniques.ipynb -> Edge Detection & Contours -> Laplacian Derivatives`

ליאור שילון

Edge Detection & Contours

Canny Edge Detection

Popular, multi-step edge detection method
Steps:

- Blur the image to reduce noise
- Compute Sobel gradients in X and Y directions
- Non-maximum suppression to keep only strong local edges
- Hysteresis thresholding to classify edge pixels

OpenCV function (Encapsulates all steps into one convenient function):

- `edges = cv2.Canny(image, min_threshold, max_threshold)`

Any gradient value larger than the maximum threshold value is considered an edge.
Any value below the minimum threshold is not considered an edge.
The gradient values in between are considered for edges according to their intensity variations.

Open: `Part2_image_processing_techniques.ipynb -> Edge Detection & Contours -> Canny Edge Detection`

ליאור שילון

Edge Detection & Contours

Contours

Contours = curves connecting points of the same intensity.
Useful for object detection, face recognition, shape analysis.

Steps to Detect Contours:

1. Convert image to grayscale
2. Binarize the image (e.g., thresholding)
3. Apply Canny edge detection
4. Use cv2.findContours() to extract contours
5. Optionally, use cv2.drawContours() to visualize them

Open: `Part2_image_processing_techniques.ipynb -> Edge Detection & Contours -> Contours`

Morphological Transformations

Morphological Transformations

Morphological transformations process images based on shapes.
They apply a structuring element (kernel) to an image, useful for removing noise, closing gaps, and extracting features!

Dilation

- Expands the boundaries of objects in an image
- Uses a structuring element (small binary matrix) to convolve the image
- A pixel becomes 1 if at least one pixel under the kernel is 1
- Causes white regions/foreground objects to grow and merge
- Makes objects more prominent and distinct
- OpenCV Function: `dilated = cv2.dilate(image, kernel, iterations=1)`
    - image → Input image (NumPy array)
    - kernel → Structuring element
    - iterations → Number of times dilation is applied

Morphological Transformations

Erosion

- Shrinks or erodes object boundaries in an image
- Uses a structuring element (kernel) to convolve the image
- A pixel remains 1 only if all pixels under the kernel are 1; otherwise it becomes 0
- Removes small details, noise, and finer structures
- Reduces thickness/size of foreground objects and white regions
- OpenCV Function: `eroded= cv2.erode(image, kernel, iterations=1)`
    - image → Input image (NumPy array)
    - kernel → Structuring element
    - iterations → Number of times erosion is applied

Morphological Transformations

Opening

- Erosion → Dilation sequence
- Removes small noise and fine objects
- Erosion eliminates small details, then dilation restores remaining objects
- Preserves the overall shape and structure of larger objects
- OpenCV Function: `opened= cv2.morphologyEx(image, cv2.MORPH_OPEN, kernel)`
    - image → Input image (NumPy array)
    - cv2.MORPH_OPEN → Specifies opening operation
    - kernel → Structuring element (convolution matrix)

Morphological Transformations

Closing

- Dilation → Erosion sequence
- Fills small gaps and holes in objects
- Dilation expands object boundaries and closes gaps
- Erosion refines boundaries and removes excessive expansion
- OpenCV Function: `closed = cv2.morphologyEx(image, cv2.MORPH_CLOSE, kernel)`
    - image → Input image (NumPy array)
    - cv2.MORPH_CLOSE → Specifies closing operation
    - kernel → Structuring element (convolution matrix)

ליאור שילון

Morphological Transformations

Morphological Gradient

- Calculates the difference between dilation and erosion of an image
- Highlights object boundaries and edges
- Useful for edge detection and emphasizing intensity transitions
- Helps in object segmentation and contour extraction
- OpenCV Function: `gradient = cv2.morphologyEx(image, cv2.MORPH_GRADIENT, kernel)`
    - image → Input image (NumPy array)
    - cv2.MORPH_GRADIENT → Specifies morphological gradient operation
    - kernel → Structuring element

ליאור שילון

Morphological Transformations

Top Hat

- Calculates the difference between the input image and its opening (image - image without small noise = only noise)
- Enhances small-scale details like fine structures or small objects
- Removes large structures or background, highlighting subtle features
- Useful for image enhancement, object detection, and feature extraction
- OpenCV Function: `tophat = cv2.morphologyEx(image, cv2.MORPH_TOPHAT, kernel)`
  - image → Input image (NumPy array)
  - cv2.MORPH_TOPHAT → Specifies top hat operation
  - kernel → Structuring element

ליאור שילון

Morphological Transformations

Black Hat

- Calculates the difference between the closing of an image and the original image (image with small gaps filled - original image = the gaps that was filled)
- Highlights larger-scale structures or regions
- Reveals features overshadowed by background
- Useful for image analysis, object detection, and text extraction
- OpenCV Function: `blackhat = cv2.morphologyEx(image, cv2.MORPH_BLACKHAT, kernel)`
  - image → Input image (NumPy array)
  - cv2.MORPH_BLACKHAT → Specifies black hat operation
  - kernel → Structuring element

Open: `Part2_image_processing_techniques.ipynb -> Morphological Transformations -> All Operations`

Template Matching

Template Matching

Template matching is used to find and locate a smaller image (template) inside a larger image.
It works by sliding the template across the target image and comparing similarity at each position.

How It Works

1. The template is moved across the target image.
2. At each location, OpenCV calculates how well the template matches that part of the image.
3. The result is a grayscale similarity map — brighter areas represent stronger matches.

Output Size

If:

- Input image = (W × H)
- Template image = (w × h)

Then the result image = (W - w + 1, H - h + 1)

Template Matching

Finding the Match

Use `cv2.minMaxLoc()` to locate:

- The maximum (or minimum) value in the result map → indicates the best match.
- That point gives the top-left corner of the detected template.

Draw a rectangle of size (w × h) from this point to visualize the matched region.

OpenCV Function Example

```
result = cv2.matchTemplate(target, template, cv2.TM_CCOEFF_NORMED)
min_val, max_val, min_loc, max_loc = cv2.minMaxLoc(result)

top_left = max_loc
bottom_right = (top_left[0] + w, top_left[1] + h)
cv2.rectangle(target, top_left, bottom_right, (0, 255, 0), 2)
```

Use Case:

- Object or pattern detection
- Logo or face matching
- Scene recognition

Open: `Part2_image_processing_techniques.ipynb -> Template Matching -> Basic Template Matching`

ליאור שילון

Part 3: Feature Extraction & Machine Learning Pipeline

Building ML-Based Computer Vision Systems

ליאור שילון

The Computer Vision Pipeline

What is the CV Pipeline

A series of components that process images through various transformations, from raw pixels to meaningful predictions. Each step builds upon the previous one to extract increasingly abstract representations

The Complete Pipeline

1. Image Ingestion: Capture and digitize images
2. Preprocessing: Standardize and enhance images
3. Feature Extraction: Extract meaningful patterns
4. Feature Selection: Choose relevant features
5. Model Training: Learn patterns from features
6. Prediction: Apply model to new images

✅ **What You'll Master:**

- **Extract powerful features using HOG, GLCM, and LBP**
- **Select the most relevant features for your model**
- **Understand the training and deployment pipeline**
- **Implement everything with practical code examples**

ליאור שילון

Feature Extraction

Definition

A feature is a measurable property of an object or event that provides distinctive information about an image

Importance

Machine learning relies heavily on features; extracting relevant and independent features is key to reliable results

Example

To differentiate cars from motorcycles, a wheel alone is insufficient. Other features like doors, roof, color, edges, and contours are also needed. Repeatable patterns across multiple samples are crucial.
Features can include: color, edges, corners, contours, angles, light intensity, etc. More distinctive features improve model quality and effectiveness

Principles for Good Features

improves feature quality, which leads to more accurate classification and better machine learning performance

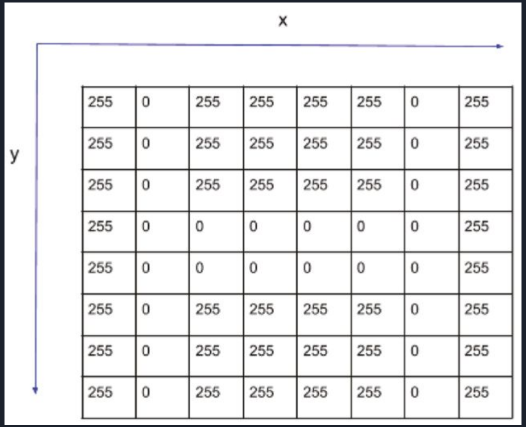
1. Distinctiveness: Should differentiate between classes.
2. Non Overlapping: Should avoid confusion between features.
3. Frequency: Should occur often enough to be reliable.
4. Robustness: Should remain consistent under varying conditions (lighting, angles, environment).
5. Identifiability: Should be easily detectable or extractable.
6. Sufficient Samples: Collect enough examples to establish reliable patterns.

ליאור שילון

How to Represent Features

Simple Example

Features extracted from an image are typically represented as a feature vector. As a simple example, consider a grayscale image, in which the features correspond to the pixel values of the image. The pixels are organized in a two-dimensional matrix, where each pixel holds a value ranging from 0 to 255.
To represent these pixel values as features, we convert them into a one-dimensional (1D) row matrix, which can be thought of as a vector or a 1D array



255	0	255	255	255	255	0	255
255	0	255	255	255	255	0	255
255	0	255	255	255	255	0	255
255	0	0	0	0	0	0	255
255	0	0	0	0	0	0	255
255	0	255	255	255	255	0	255
255	0	255	255	255	255	0	255
255	0	255	255	255	255	0	255

[row 1 pixels, row 2 pixels, …, row n pixels]
OR
[255,0,255,255,255,0,255,....255,0,255,255,255,0,255]

Feature Extraction for Machine Learning

- **Traditional ML:**
  Most machine learning algorithms require features to be manually extracted from images and then fed into the model for training.
- **Deep Learning:**
  Certain deep learning algorithms, such as Convolutional Neural Networks (CNNs), can automatically extract features during training.

ליאור שילון

Feature Extraction Methods

Color Histograms

1. Definition: A representation of the distribution of colors in an image.
   It counts how many pixels have colors in specific ranges, providing a statistical summary of the image's color content.
   - X-axis: Pixel values or ranges (0–255 for 8-bit images).
   - Y-axis: Frequency or count of pixels for each value.
   - The peak indicates the most frequently occurring color.
2. Binning:
   - Instead of plotting all 256 pixel values, we can group them into bins.
   - Example: 8 bins, each covering 32 pixel values.
   - The height of each bin is the sum of pixels in that range.
3. Color Images:
   - RGB images have three channels (Red, Green, Blue).
   - Plotting a separate histogram for each channel helps analyze the intensity distribution of colors.
   - Histograms can be used as features in machine learning algorithms.
4. Applications:
   - Image Analysis: Understanding color, contrast, and brightness.
   - Image Enhancement: Techniques like histogram equalization improve image quality based on pixel intensity distribution.

ליאור שילון

Feature Extraction Methods

How to Calculate a Histogram

OpenCV simplifies the process of calculating a histogram by providing a user friendly function called calcHist():

```
cv2.calcHist(images, channels, mask, histSize, ranges, accumulate)
```

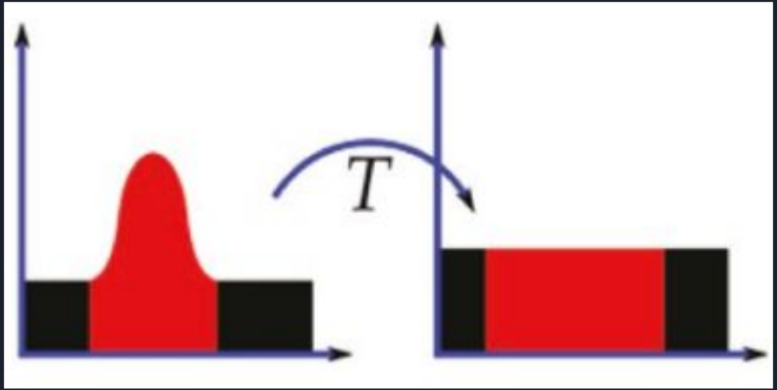
- `images`: This is a NumPy array of image pixels. If you have only one image, just wrap the NumPy variable within a pair of square brackets, e.g., [image].
- `channels`: This is an array of indexes of channels we want to calculate the histogram for. This will be [0] for grayscale images and [0,1,2] for RGB color images.
- `mask`: This is an optional argument. If you do not supply a mask, the histogram will be calculated for all the pixels in the image or images. If you supply a mask, the histogram will be calculated for the masked pixels only. (Remember masks from previous chapter).
- `histSize`: This is the number of bins. If we pass this value as [64,64,64], this means that each channel will have 64 bins. The bin size can vary for different channels.
- `ranges`: This is the range of pixel values, which is normally [0,255] for grayscale and RGB color images. This value may be different in other color schemes, but for now, let's stick to RGB only.
- `accumulate`: This is the accumulation flag. If it is set, the histogram is not cleared in the beginning when it is allocated. This feature enables you to compute a single histogram from several sets of arrays or to update the histogram in time. The default value is None

Open: `Part3_feature_extraction.ipynb -> Histogram`
`-> Grayscale Histogram/RGB Color Histogram`

Feature Extraction Methods

Histogram Equalizer

- Enhances image contrast using pixel intensity redistribution.
- Balances bright and dark areas by spreading out intensity values.
- Makes details in both light and dark regions more visible



```
equalizedImage = cv2.equalizeHist(image)
```

Open: `Part3_feature_extraction.ipynb -> Histogram -> Histogram Equalizer`

ליאור שילון

Feature Extraction Methods

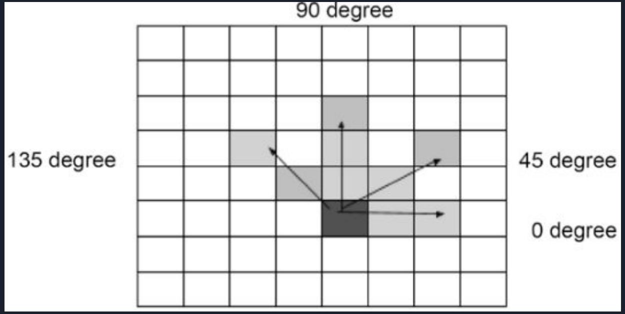
GLCM

- The Gray-Level Co-occurrence Matrix captures how often pairs of pixels with specific values and spatial relationships occur in an image. It's a powerful tool for texture analysis.
- Used for grayscale images to analyze spatial relationships between pixels.
- Common angles: 0° (horizontal), 90° (vertical), 45°, 135° (diagonal).
- Unlike histograms, GLCM includes texture and spatial information.
- Key texture features (e.g., contrast, energy, homogeneity) are derived from GLCM for machine learning.
- Calculated using skimage.feature (not directly available in OpenCV)

How GLCM Works

For each pixel pair at distance d and angle θ:

```
GLCM[i,j] = Count of pixel pairs where first pixel = i and second pixel = j
```



M GLCM

$$\begin{bmatrix}0 & 1\\1 & 0\end{bmatrix} \xrightarrow{\theta=0\,,\, d=1} \begin{bmatrix}0 & 1\\1 & 0\end{bmatrix}$$

Algorithm Example: link

ליאור שילון

Feature Extraction Methods

GLCM - Function

```
graycomatrix(image, distances, angles, levels, symmetric,normed)
```

- `image`:
  This is the NumPy representation of a grayscale image. Remember, the image must be grayscale.
- `distances`:
  This is a list of pixel-pair distance offsets.
- `angles`:
  This is a list of angles between the pair of pixels. Make sure the angle is a radian and not a degree.
- `levels`:
  This is an optional parameter and meant for images having 16-bit pixel values. In most cases, we use 8-bit image pixels that can have values ranging from 0 to 255. For an 8-bit image, the max value for this parameter is 256.
- `symmetric`:
  This is an optional parameter and takes a Boolean. The value True means the output matrix will be symmetric. The default is False.
- `normed`:
  This is also an optional parameter that takes a Boolean. The Boolean True means that each output matrix is normalized by dividing by the total number of accumulated co-occurrences for the given offset. The default is False.

`Returns`: The graycomatrix() function returns a 4D ndarray. This is the gray-level co-occurrence histogram.
The output value P[i,j,d,theta] represents how many times the gray-level j occurs at a distance d and angle theta from the gray-level j.
If the parameter normed is False (which is the default), the output is of type uint32 (a 32-bit unsigned integer); otherwise, it is float64 (a 64-bit floating point).

Open: `Part3_feature_extraction.ipynb ->GLCM-> GLCM Calculation`

ליאור שילון

Feature Extraction Methods

GLCM - Uses

the GLCM is not directly used as a feature, but we use this to calculate some useful statistics, which gives us an idea about the texture of the image.
The following table lists the statistics we can derive

Statistic	Description	Formula
Contrast	Measures the local variations in the GLCM.	$\sum_{i,j=0}^{levels-1} P_{i,j}(i-j)^2$
Correlation	Measures the joint probability occurrence of the specified pixel pairs	$\sum_{i,j=0}^{levels-1} P_{i,j}\left[\frac{(i-\mu_i)(j-\mu_j)}{\sqrt{(\sigma_i^2)(\sigma_j^2)}}\right]$
Energy	Provides the sum of squared elements in the GLCM. Also known as uniformity or the angular second moment	$\sqrt{ASM}$ $\quad \text{ASM} = \sum_{i,j=0}^{levels-1} P_{i,j}^2$
Homogeneity	Measures the closeness of the distribution of elements in the GLCM to the GLCM diagonal.	$\sum_{i,j=0}^{levels-1} \frac{P_{i,j}}{1+(i-j)^2}$
Dissimilarity	Measures the difference between pairs of pixel intensities in the GLCM	$\sum_{i,j=0}^{levels-1} P_{i,j}\lvert i-j\rvert$

Feature Extraction Methods

HOGs

Histograms of oriented gradients (HOGs) captures the distribution of gradient orientations in localized portions of an image. It's particularly effective for detecting objects with distinctive shapes, like pedestrians or vehicles.

The HOG Algorithm Pipeline

1. Global Image Normalization (optional)
   - Reduces illumination effects.
   - Common methods:
     i. Square-root normalization: preferred ($\sqrt{p}$).
     ii. Variance normalization: $(p - \mu)/\sigma$ gives best results.
2. Compute Gradients (G_x, G_y)
   - Captures edges, contours, and textures.
   - Gradient magnitude: $|G| = \sqrt{(G_x^2 + G_y^2)}$
   - Gradient orientation: $\Theta = \arctan(G_y / G_x)$
3. Compute Gradient Histograms
   - Image divided into cells.
   - For each cell, a histogram of orientations is built.
   - Each pixel votes for an orientation bin weighted by its gradient magnitude.
4. Normalize Across Blocks
   - Group neighboring cells into blocks.
   - Normalize histograms within each block to reduce lighting/shadow effects.
   - These normalized descriptors are the HOG features.
5. Flatten into Feature Vector
   - Concatenate all block histograms into a single feature vector for machine learning or object detection models.

Feature Extraction Methods

HOGs - Function

```
out, hog_image = hog(image, orientations=9, pixels_per_cell=(8, 8), cells_per_block=(3, 3),
block_norm='L2-Hys', visualize=False,transform_sqrt=False, feature_vector=True, multichannel=None)
```

- `image`: This is the NumPy representation of the input image.
- `orientations`: The number of orientation bins defaults to 9.
- `pixels_per_cell`: This is the number of pixels in each cell as a tuple. it defaults to (8,8) for an 8×8 cell size.
- `cells_per_block`: This is the number of cells in each block, as a tuple. it defaults to (3,3), which is for 3×3 cells, not pixels.
- `block_norm`: This is the block normalization method as a string with one of these values: L1, L1-sqrt, L2, L2-Hys. These normalization strings are explained here:

$$\text{L1-norm} = \sum_{r=1}^{n} |X_r|$$

$$\text{L1-sqrt} = \sqrt{\sum_{r=1}^{n} |X_r|}$$

$$- \quad \text{L2-norm} = \sqrt{\sum_{r=1}^{n} |X_r|^2}$$

L2-Hys: This is the default normalization for the parameter block_norm.
L2-Hys is calculated by first taking the L2-normalization, limiting the result to a maximum of 0.2, and then recalculating the L2-normalization

Feature Extraction Methods

HOGs - Function

- `visualize`: If this is set to True, the function also returns an image of the HOG. Its default value is set to False.
- `transform_sqrt`: If set to True, the function will apply power law compression to normalize the image before processing.
- `feature_vector`: The default value of this argument is set to True, which instructs the function to return the output data as a feature vector.
- `multichannel`: Set the value of this argument to True to indicate the input image contains multichannels. The dimensions of an image are generally represented as height × width × channel. If the value of this argument is True, the last dimension (channel) is interpreted as the color channel, otherwise as spatial

The function returns:

- `out`: The function returns an ndarray containing (n_blocks_row, n_blocks_col, n_cells_row, n_cells_col, n_orient). This is the HOG descriptor for the image. If the argument feature_vector is True, a 1D (flattened) array is returned.
- `hog_image`: If the argument visualize is set to True, the function also returns a visualization of the HOG image.

Feature Extraction Methods

HOGs - Example

Open: `Part3_feature_extraction.ipynb ->HOGs`

Algorithm Example: link

Mathematical Foundation

$$|G| = \sqrt{G_x^2 + G_y^2}$$

$$\Theta = \arctan\left(G_y / G_x\right)$$

⚠️ It is worth mentioning that the hog() function generates a histogram of very high dimensionality. A 32×32 image with pixel_per_cell=(4,4) and cells_per_block=(2,2) will generate 1,764-dimension results. Similarly, a 128×128 pixel image will generate 34,596-dimension output. It is, therefore, extremely important to pay attention

Feature Extraction Methods

LBP

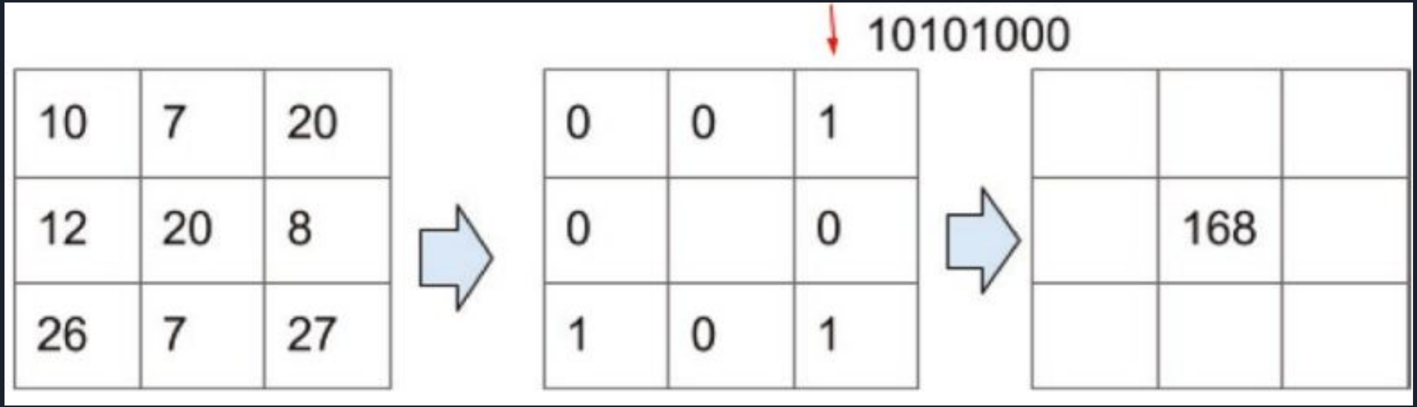
Local Binary Patterns (LBP) serves as a feature descriptor utilized in the classification of image textures.
Captures texture by comparing each pixel with its neighbors.

The LBP Algorithm Pipeline

1. Compare Neighboring Pixels
   - For each pixel, compare its 8 neighbors (3x3 window):
     1. Neighbor < central pixel → 0
     2. Neighbor ≥ central pixel → 1
2. Form Binary Number
   - Start from a fixed neighbor (e.g., top-right) and move clockwise.
   - Concatenate the 0s and 1s to form an 8-bit binary number.
   - Convert binary → decimal → new value for the central pixel.
3. Repeat for All Pixels
   - Apply the same comparison and binary conversion for every pixel.
4. Create LBP Array
   - Arrange the new pixel values into a matrix (LBP array).
5. Compute Histogram
   - Calculate a histogram over the LBP array.
   - The histogram is the LBP feature vector for texture analysis.

Feature Extraction Methods

LBP - Example



The LBP Problem

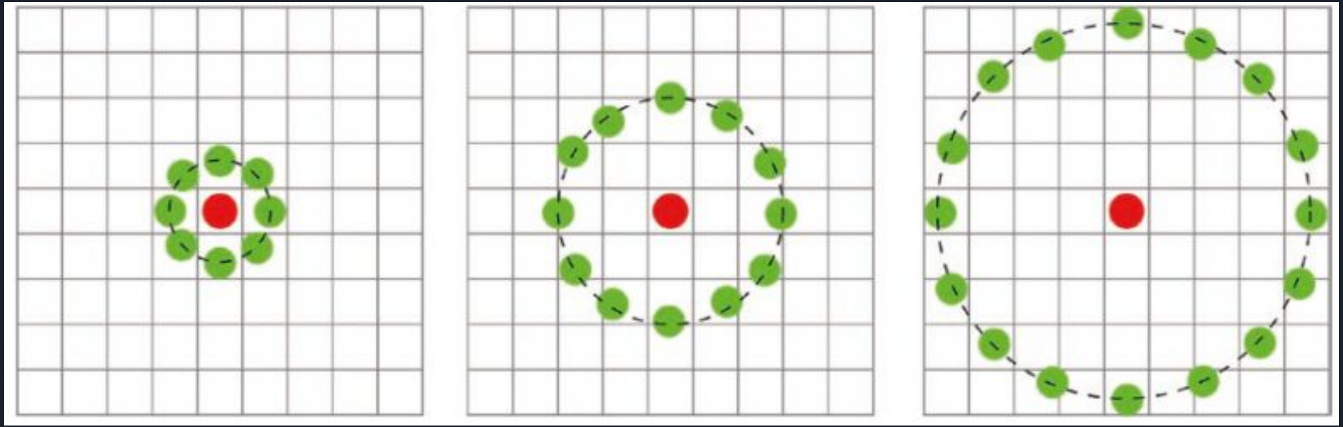
This approach of calculating an LBP feature vector allows us to capture finer details of the image texture. But for most machine learning classification problems, fine-grained features may not give the desired outcome, especially when the input images are of varying scales of texture.

Feature Extraction Methods

The Solution

To overcome this problem, we have an enhanced version of LBP that allows for variable neighborhood sizes. It gives us two additional parameters to work with:

- Instead of a fixed square neighborhood, we can define the number of points, p, in a circularly symmetric neighborhood.
- The radius of the circle, r, allows us to define different neighborhood sizes.



Feature Extraction Methods

LBP - Function

```
sk.local_binary_pattern(image=image, P=p, R=r, method='default')
```

- `image`: The NumPy representation of a grayscale image.
- `P`: The number of neighborhood points along the circle surrounding the point for which LBP is being calculated. (This is the number of green dots in the previous slide image).
- `R`: This is a floating-point number and defines the radius of the circle.
- `method`: This parameter takes any of the following string values:
  - `default`: This instructs the function to calculate original LBP based on grayscale without considering the rotation invariant (description of a rotationally invariant binary descriptor).
  - `ror`: This method instructs the function to use a rotationally invariant binary descriptor.
  - `uniform`: This uses an improved rotation invariance with uniform patterns and finer quantization of the angular space, which is grayscale and rotation invariant.
  - `nri_uniform`: Non-rotation-invariant uniform patterns variant, which is only grayscale invariant.
  - `var`: Rotation invariant variance measures of the contrast of local image texture, which is rotation but not grayscale invariant.
  -

The output of the function local_binary_pattern() is an ndarray representing an LBP image

Open: `Part3_feature_extraction.ipynb ->LBP`

Feature Selection Strategies

Feature Selection

The process of selecting relevant features and eliminating irrelevant or redundant ones to create an optimal subset for model training.

Why Feature Selection Matters

- Reduces model complexity and improves interpretability
- Decreases training time
- Enhances model accuracy
- Prevents overfitting and improves generalization

Key Concept

There's an optimal number of features - too few or too many degrades performance.

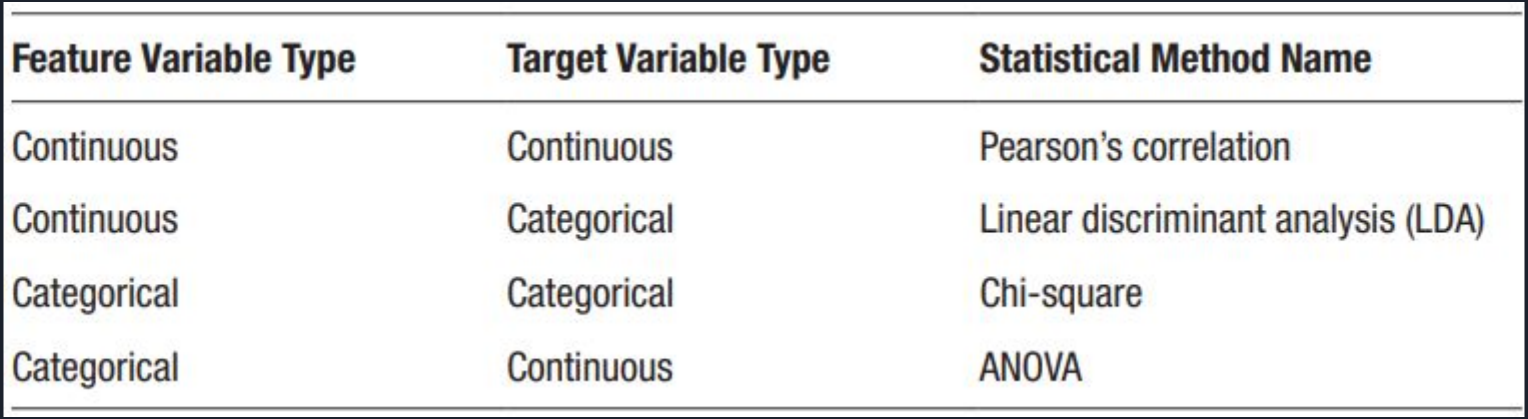
Feature Selection vs. Feature Extraction

- Extraction: Creates new features from existing data
- Selection: Chooses subset of existing features

Feature Selection Strategies

Feature Selection - Filter Method

- Pre-processing approach - selects features before training
- Based on statistical correlation between features and target
- Independent of machine learning algorithm
- Uses statistical tests: Pearson's correlation, Chi-square, ANOVA, LDA



Feature Variable Type	Target Variable Type	Statistical Method Name
Continuous	Continuous	Pearson’s correlation
Continuous	Categorical	Linear discriminant analysis (LDA)
Categorical	Categorical	Chi-square
Categorical	Continuous	ANOVA

Feature Selection Strategies

Feature Selection - Wrapper Method

- Trial-and-error approach - trains models with different feature subsets
- Evaluates model performance to add/remove features
- Computationally expensive
- Techniques: Forward selection, Backward elimination, Recursive feature elimination

Technique	Description
Forward selection	Start with one feature and build and evaluate the model. Iteratively add features that best improve the model.
Backward elimination	Start with all features and build and evaluate the model. Iterate through by eliminating features until you get the best model. Repeat this until no improvement is observed on feature removal.
Recursive feature elimination	Repeatedly create models and set aside the best- or worst-performing feature at each iteration. Rank the features either by their coefficients or by feature importance, and eliminate the least important features. Recursively create new models with the leftover features until all features are exhausted.

Feature Selection Strategies

Feature Selection - Embedded Method

- Feature selection happens during model training
- Algorithm determines important features automatically
- Examples: LASSO (L1), Ridge (L2) regularization
- Most efficient - least expensive computationally

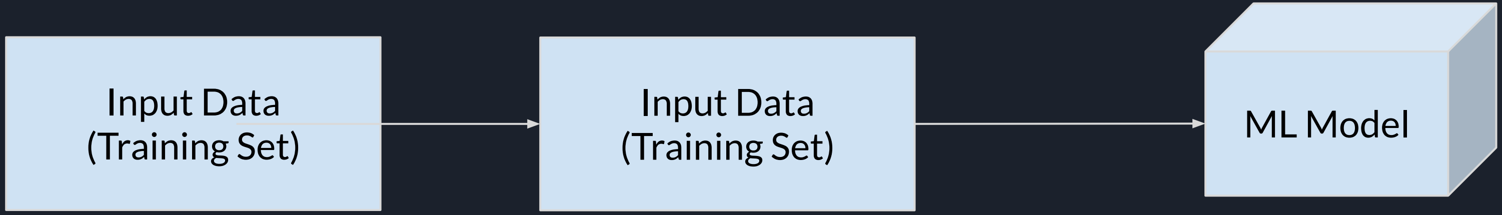
regularization	Description
Lasso regression	uses L1 regularization and adds a penalty equivalent to the absolute value of the magnitude of coefficients
Ridge regression	uses L2 regularization and adds a penalty equivalent to the square of the magnitude of coefficients

Open: `Part3_feature_selection.ipynb`

Model Training and Deployment

Model Training

- Algorithms use the extracted features to identify patterns.
- The resulting mathematical representation is called a model.
- The process of creating this model from the training set is called model training.



Machine Learning Paradigms We Will Discuss

- Supervised Learning
- Unsupervised Learning

Model Training and Deployment

Supervised Learning with Image Features

Feature representation:

- An 8×8 image → 64 pixels → 64 features per image.
- Each image's features are arranged as a vector (row).
- The feature set has one row per image.

Labels:

- Each training image has a known class (e.g., dog = 0, cat = 1).
- Labels are numeric to be used in machine learning.
- Labels are also called target variables or dependent variables.

Training process:

- Feed the feature vectors (X) and labels (y) to a learning algorithm.
- The algorithm learns a function that maps features → labels.
- This process is called supervised learning.

Goal:

- Train a model that can classify new input images based on learned patterns.

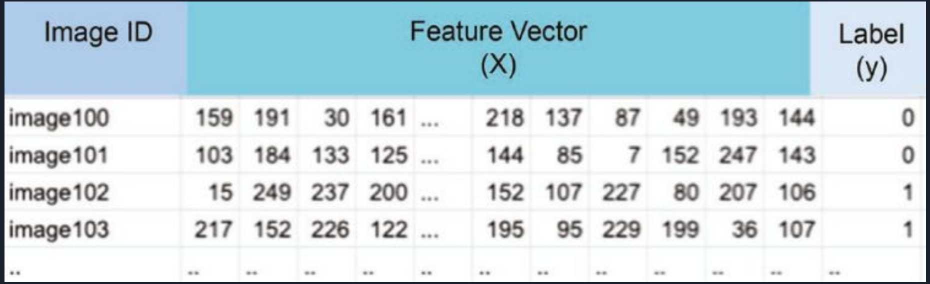


Image ID	Feature Vector (X)											Label (y)
image100	159	191	30	161	...	218	137	87	49	193	144	0
image101	103	184	133	125	...	144	85	7	152	247	143	0
image102	15	249	237	200	...	152	107	227	80	207	106	1
image103	217	152	226	122	...	195	95	229	199	36	107	1
..	..	..	..	..	..	..	..	..	..	..	..	..

Model Training and Deployment

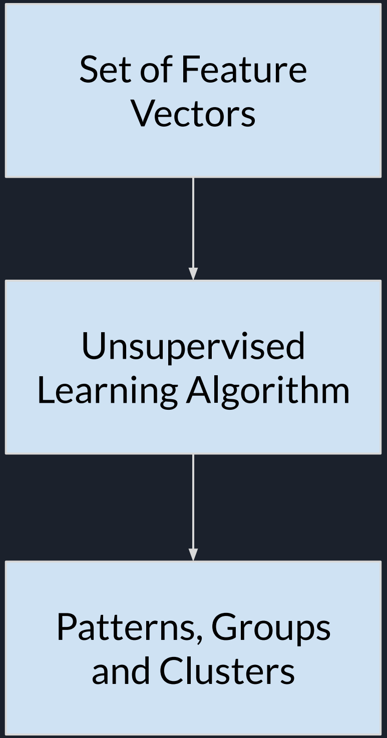
Unsupervised Learning with Image Features

Definition:
- **Learning from feature vectors without labels.**

Goal:
- **Identify patterns, structures, or similarities in the data.**

Example task:
- **Group or cluster images based on feature similarity.**

Uses:
- **Clustering and grouping of data**
- **Can help generate labels for supervised learning**


Set of Feature Vectors
Unsupervised Learning Algorithm
Patterns, Groups and Clusters

Model Training and Deployment

Model Deployment and Utilization

Purpose of a trained model:

- Supervised learning: Predict outcomes (classes or continuous values) from input features.
- Unsupervised learning: Assign input features to groups or clusters, which can also be used to generate labels for supervised learning.

Deployment:

- Make the model available for real-time or batch predictions on new input data.
- Input data undergoes the same feature extraction process used during training.
- Once a model achieves acceptable accuracy, it is versioned and deployed.
  Retraining occurs only if accuracy drops or new data improves performance.
- Models are often used more frequently than they are retrained, potentially processing hundreds to millions of images depending on the application.

Model Training and Deployment

Deployment architectures

Deployment Type	Description	Pros	Cons / Considerations
Embedded model	Model artifacts included in application code. deployed as an internal library function.	Good for edge/IoT applications	Hard to update; not suitable for large-scale enterprise data. need to rebuild app for new model versions
Separate service	Model wrapped in an independent service. applications call it remotely.	Can update/redeploy independently. separation of concerns	Remote calls may introduce latency
RESTful web service	Model accessed via REST API over TCP/IP.	Scalable. supports load balancing	Network latency may affect performance
Distributed processing	Model deployed across cluster nodes. input data stored in distributed storage (HDFS, S3, etc.).	Highly scalable. parallel processing	Requires distributed infrastructure and coordination