Introduction to Computer Vision: Exploring How Machines Learn to See

This presentation explores the fascinating world of computer vision, where algorithms enable computers to interpret and understand visual information from the world around us. From basic image processing to advanced feature detection techniques, we'll examine the core concepts that allow machines to "see" and make sense of visual data.

Why Computer Vision Matters

The Power of Visual Data

Computer vision transforms how machines interact with our visual world:

- Humans process ~80% of information visually
- Digital images/videos generate petabytes of untapped data daily

- CV enables automation of visual tasks previously requiring human intelligence
- Powers innovations from facial recognition to autonomous vehicles

- Market projected to reach $48.6 billion by 2027

Applications Transforming Industries

Healthcare

Medical image analysis for disease detection, surgical assistance, patient monitoring, and diagnostic support across radiology, pathology, and ophthalmology

Manufacturing

Defect detection, quality control, inventory management, and robotic guidance systems that increase production efficiency and reduce errors

Transportation

Autonomous vehicles, traffic monitoring, license plate recognition, and driver assistance systems that enhance safety and efficiency

Retail

Cashierless stores, inventory tracking, customer behavior analysis, and personalized shopping experiences

Computer Vision Pipeline

Image Acquisition

Capturing visual data through cameras, sensors, or importing existing images

Pre-processing

Enhancing images through filtering, normalization, and noise reduction

Feature Extraction

Identifying key points, edges, and descriptive elements

Decision Making

Classification, object detection, and semantic understanding

Image Processing: The Foundation

Image processing forms the backbone of computer vision by transforming raw pixel data into more useful forms for analysis:

- Converts visual information into numerical data computers can process
- Prepares images for feature extraction and recognition tasks
- Enhances image quality and removes unwanted artifacts
- Reduces dimensionality to focus on relevant information
- Enables consistent interpretation despite varying lighting, angles, and noise

Reading & Displaying Images

Understanding Digital Images

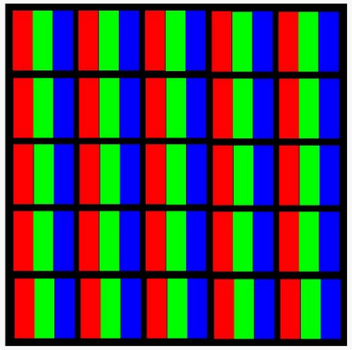
Images are represented as multi-dimensional arrays of pixel values:

- Grayscale: 2D array with intensity values (0-255)
- Color: 3D array with channels (typically RGB)
- Alpha channel: Optional 4th dimension for transparency

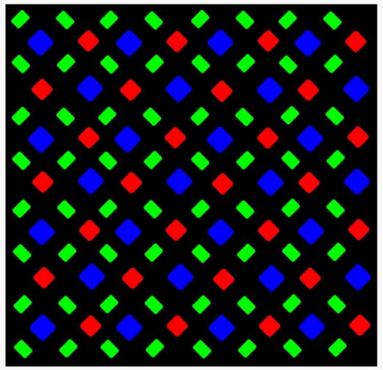
Common libraries like OpenCV make image manipulation straightforward:

```
import cv2# Read an imageimg = cv2.imread('image.jpg')#
Display the imagecv2.imshow('Image', img)cv2.waitKey(0)
```

Highly enlarged schematic representation of the arrangement of subpixels



5 x 5 pixel with RGB subpixels

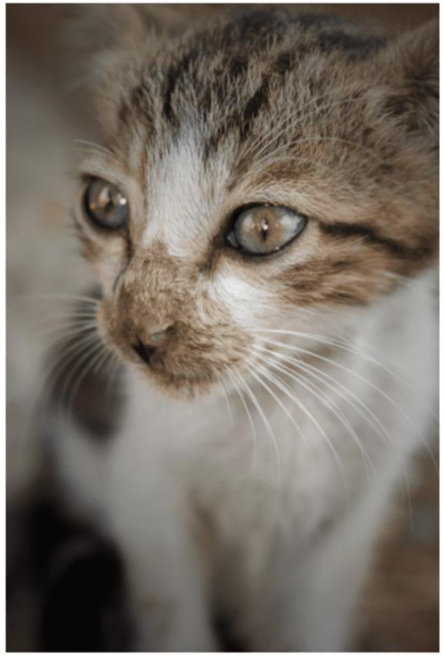


Subpixel layout of an iPhone

IONOS

Grayscale Conversion: Simplifying Visual Data

Original RGB Image



Grayscale Image

Why Convert to Grayscale?

- Reduces computational complexity (1 channel vs 3)
- Simplifies algorithms for edge detection and thresholding
- Preserves important structural information
- Reduces sensitivity to color variations

Common Conversion Method:

```
# Convert to grayscalegray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

The weighted conversion typically uses: Y = 0.299R + 0.587G + 0.114B

Blurring & Smoothing: Noise Reduction

Gaussian Blur

Uses a Gaussian distribution to create weights for averaging. Effective for natural noise patterns and preserving edges better than simple averaging.

```
blur = cv2.GaussianBlur(img, (5,5), 0)
```

Median Blur

Replaces each pixel with the median of neighboring pixels. Excellent for salt-and-pepper noise while preserving edges.

```
median = cv2.medianBlur(img, 5)
```

Bilateral Filter

Preserves edges while blurring by considering both spatial proximity and intensity differences.

```
bilateral =
cv2.bilateralFilter(img, 9, 75, 75)
```

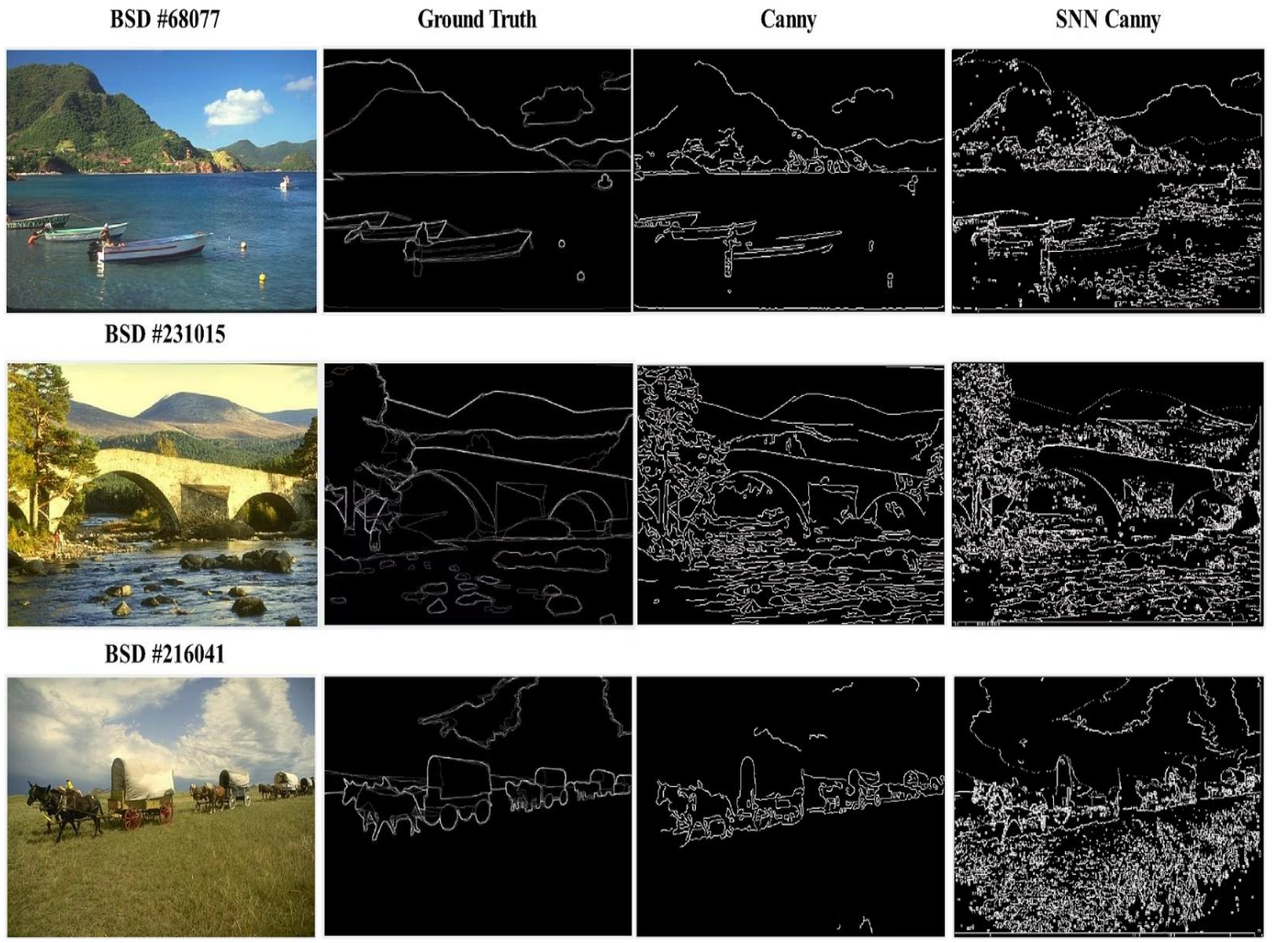
Edge Detection: Finding Boundaries

Edge detection identifies sudden changes in pixel intensity that often represent object boundaries—crucial for shape analysis and object recognition.

Common Operators:

- **Sobel:** Emphasizes horizontal or vertical edges using gradient-based methods
- **Laplacian:** Detects edges using a second derivative measure
- **Canny:** Multi-stage algorithm with noise reduction, gradient calculation, non-maximum suppression, and hysteresis thresholding

```
# Canny edge detectionedges = cv2.Canny(gray, 100, 200)
```



Thresholding: Binary Image Creation

Original Image

Otsu Thresholding

Adaptive Mean Thresholding

Adaptive Gaussian Thresholding

Separating Foreground from Background

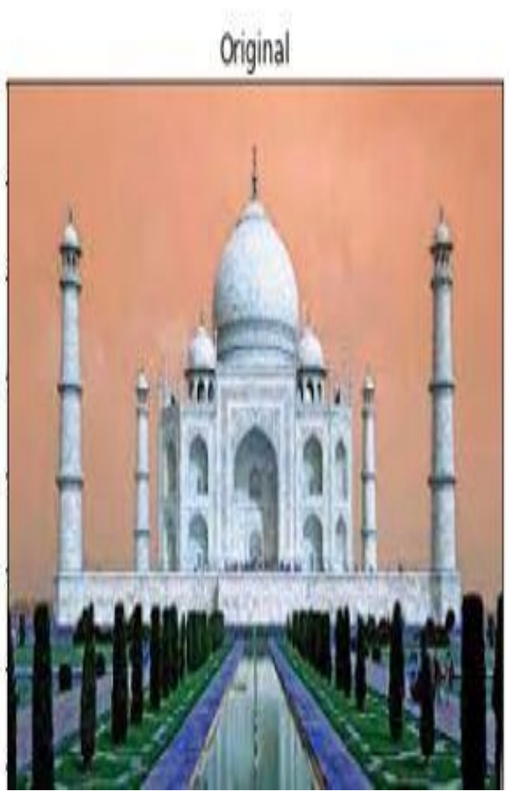
Thresholding converts grayscale images to binary by classifying pixels as either foreground or background based on intensity values.

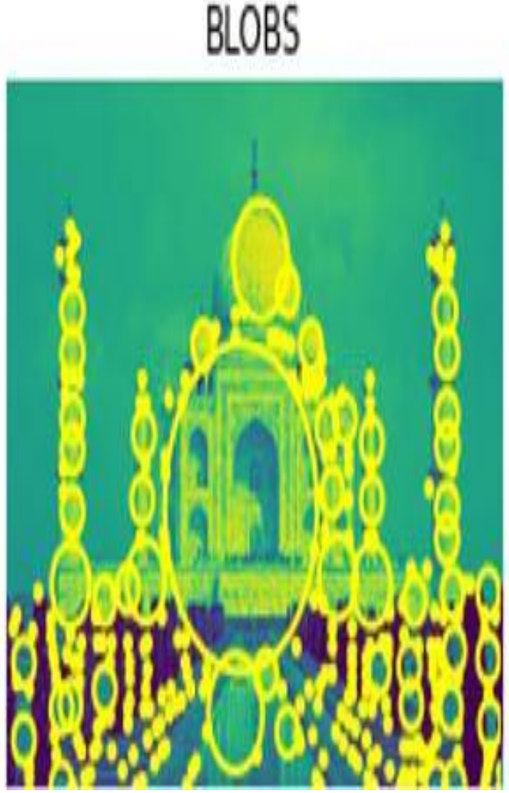
Types of Thresholding:

- **Simple/Binary:** Fixed threshold value (global)
- **Adaptive:** Threshold varies across image regions
- **Otsu's:** Automatically determines optimal threshold

```
# Simple thresholdingret, thresh = cv2.threshold(gray, 127, 255,  cv2.THRESH_BINARY)# Adaptive thresholdingadaptive = cv2.adaptiveThreshold(gray, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY, 11, 2)
```

Feature Detection: Beyond Pixels





Feature detection identifies distinctive points or regions that:

Remain stable under transformations

Good features are recognizable despite changes in scale, rotation, perspective, and lighting conditions

Are highly distinctive

Each feature should be uniquely identifiable from others in the same image

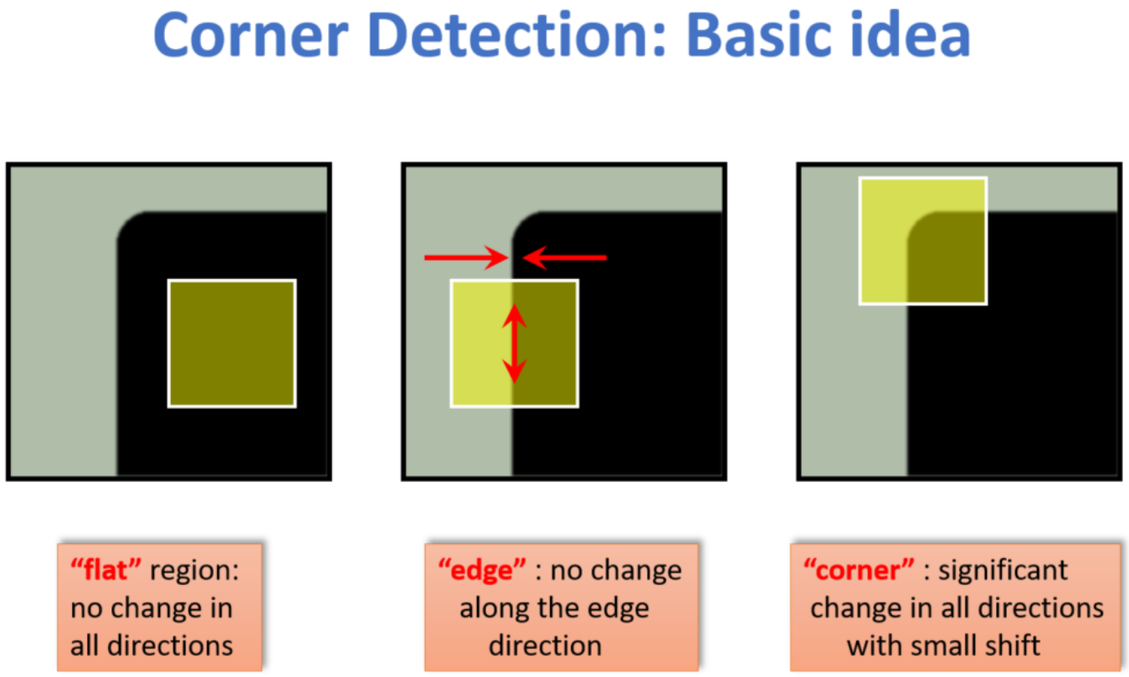
Form compact representations

Allow efficient storage and matching operations even for complex scenes

Enable higher-level vision tasks

Support object recognition, image stitching, 3D reconstruction, and motion tracking

Harris Corner Detection



Finding Corners in Images

Harris corner detector identifies points where intensity changes in multiple directions simultaneously—typically corners.

How It Works:

1. Compute x and y derivatives of the image
2. Create the Harris matrix M for each pixel
3. Calculate corner response R using determinant and trace
4. Apply non-maximum suppression
5. Threshold to identify strong corners

```
# Harris corner detectiongray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)corners = cv2.cornerHarris(gray, 2, 3, 0.04)
```

Pros: Rotation invariant**Cons:** Not scale invariant

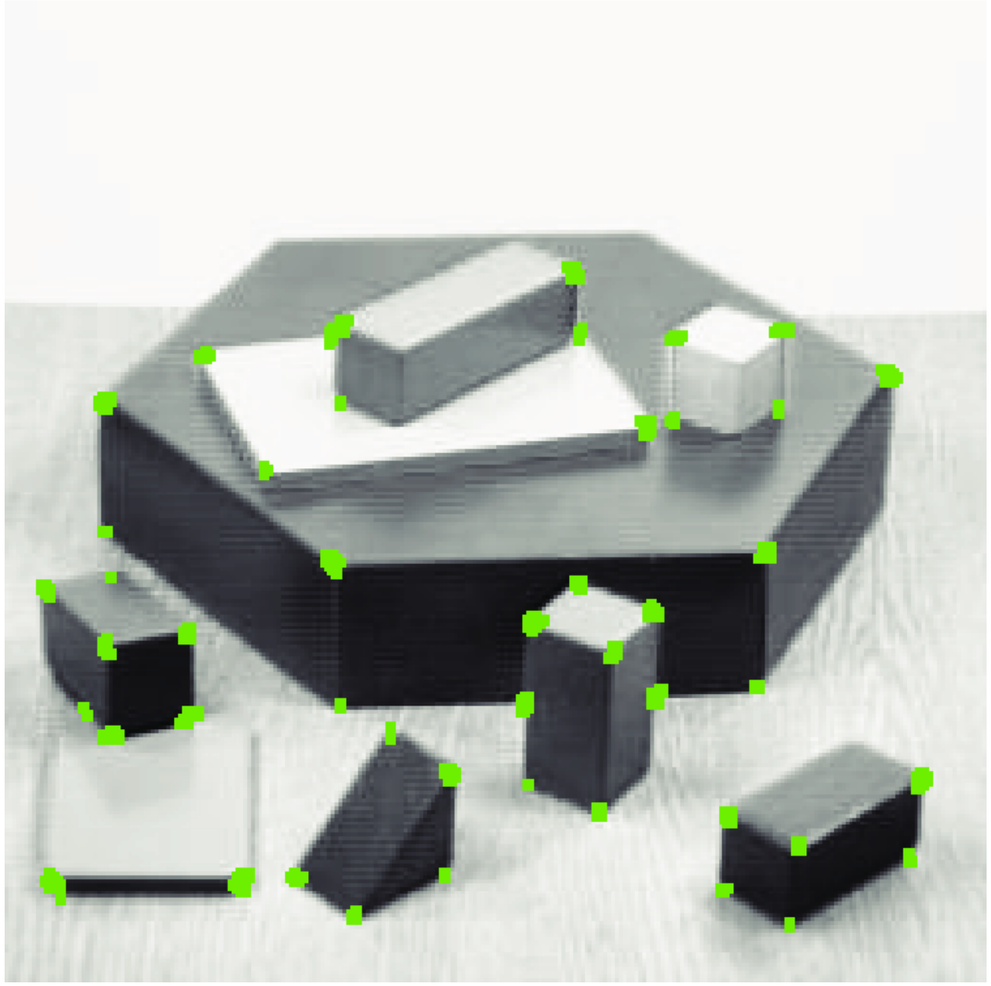
Shi-Tomasi Corner Detection

Good Features to Track

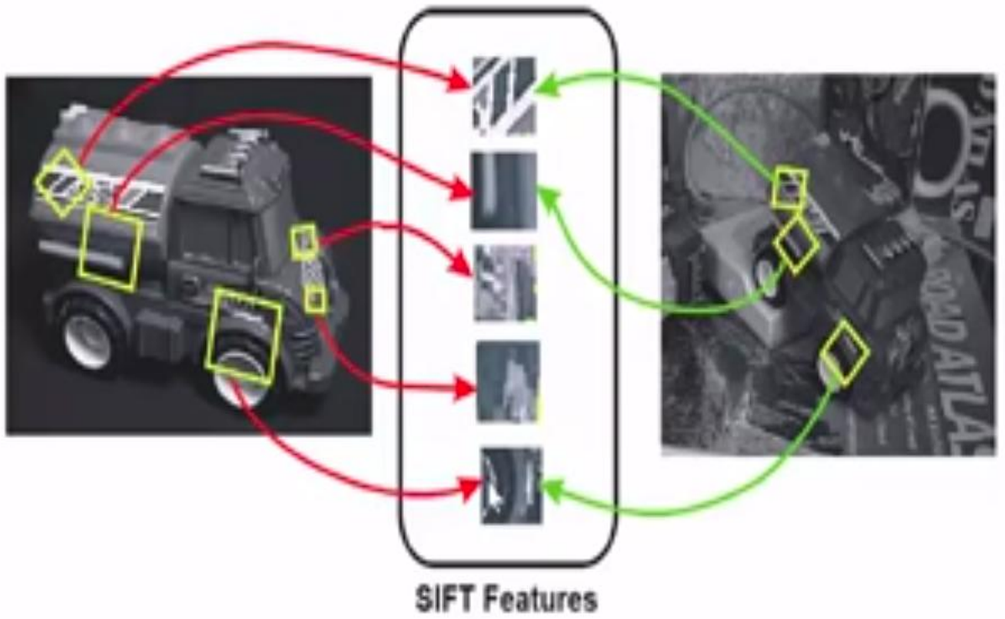
An improvement over Harris corner detection that's especially useful for tracking applications:

- Modifies the Harris corner response calculation
- Uses minimum eigenvalue rather than the Harris R function
- Provides more stable and reliable corner detection
- Returns exactly N strongest corners, sorted by quality
- Ensures minimum distance between detected points
- Widely used in optical flow and motion tracking

```
# Shi-Tomasi corner detectioncorners =
cv2.goodFeaturesToTrack(    gray, maxCorners=25,
qualityLevel=0.01, minDistance=10)
```



SIFT: Scale-Invariant Feature Transform



Scale Invariance

Detects features across multiple scales using Difference of Gaussian (DoG) pyramid, making it robust to zooming and size changes

Rotation Invariance

Assigns orientation to keypoints based on local gradient directions, enabling recognition regardless of rotation

Distinctive Descriptors

Creates 128-dimensional feature vectors using gradient histograms that uniquely identify keypoints

```
# SIFT implementationsift = cv2.SIFT_create()keypoints, descriptors = sift.detectAndCompute(gray, None)
```

Note: While SIFT offers excellent performance, it's patented technology. OpenCV moved it to the "contrib" modules, and some applications use alternatives like ORB for licensing reasons.

ORB: Oriented FAST and Rotated BRIEF

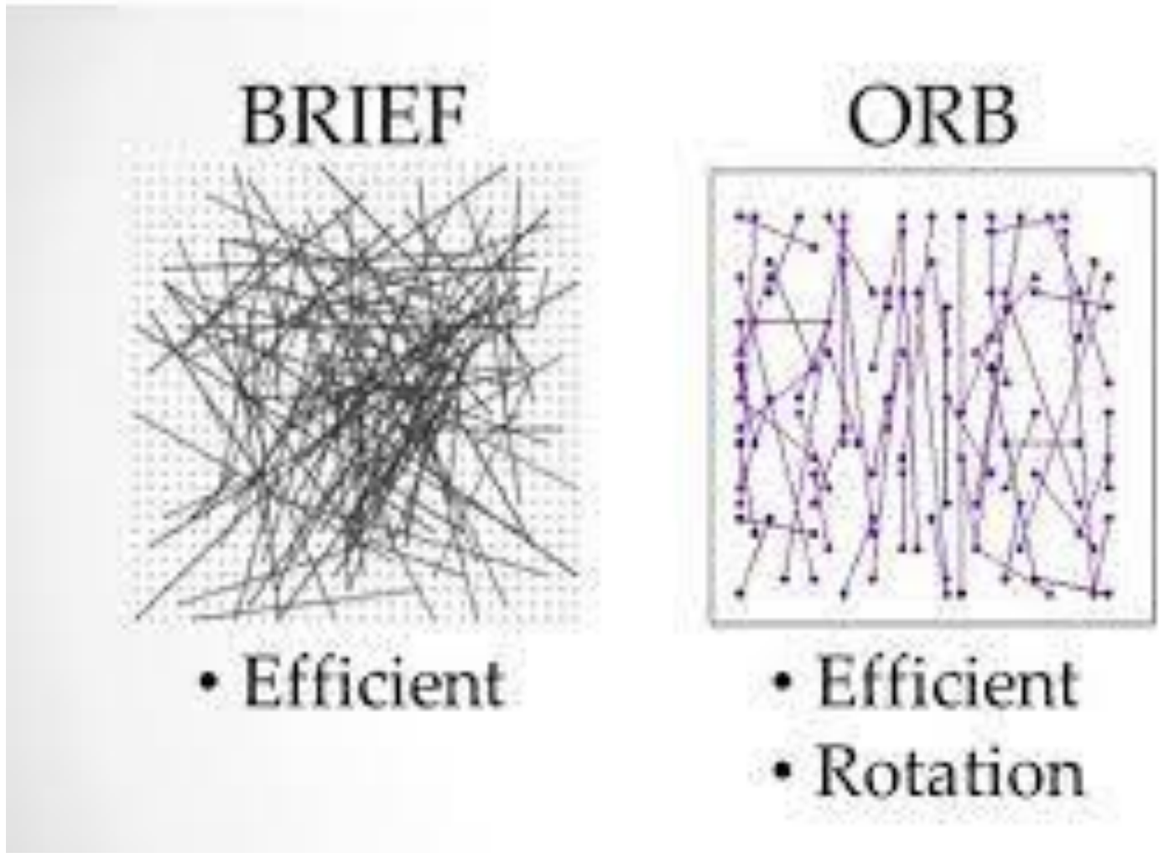
A Free Alternative to SIFT/SURF

ORB combines modified versions of FAST keypoint detector and BRIEF descriptor:

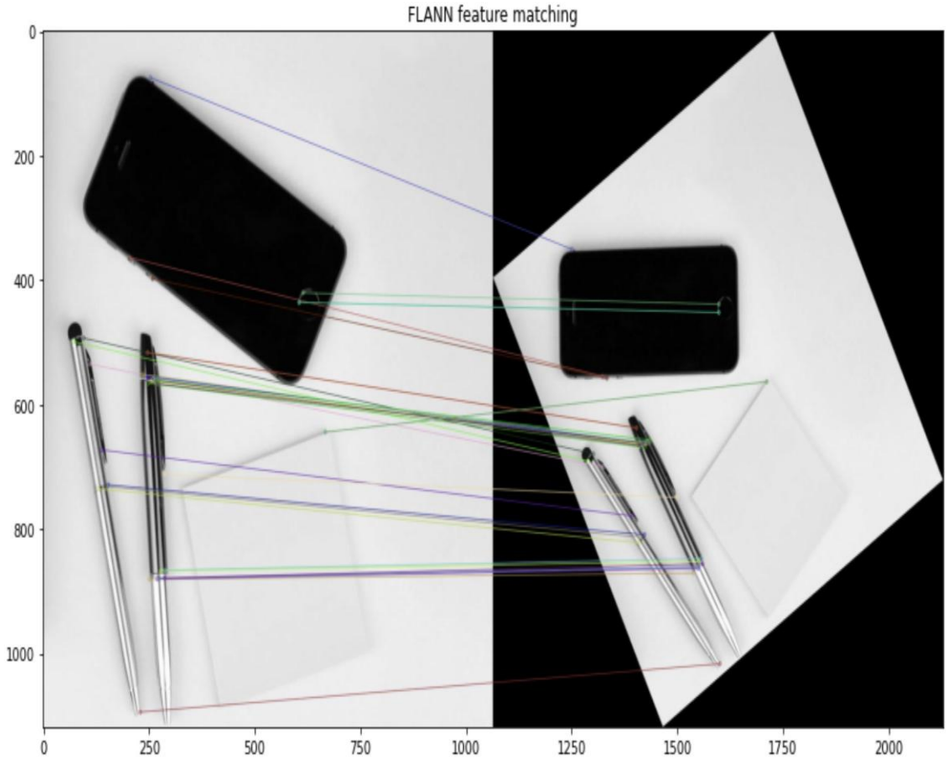
- **FAST:** Finds keypoints by comparing pixel intensities in a circular pattern

- Adds orientation computation using intensity centroid
- **BRIEF:** Creates binary descriptor through intensity comparisons
- Rotates BRIEF pattern according to keypoint orientation
- Computationally efficient (much faster than SIFT)
- Free to use commercially (no patents)

```
# ORB implementationorb = cv2.ORB_create()keypoints,
descriptors = orb.detectAndCompute(gray, None)
```



Feature Matching: Finding Correspondences



Connecting Features Across Images

Feature matching identifies corresponding points between images—essential for image stitching, object recognition, and 3D reconstruction.

Brute Force Matching

Compares each descriptor in first set with all descriptors in second set using distance measures (Euclidean, Hamming, etc.)

```
bf = cv2.BFMatcher()matches = bf.knnMatch(desc1, desc2, k=2)
```

FLANN Matching

Fast Library for Approximate Nearest Neighbors provides faster matching for large datasets through optimized algorithms

```
flann = cv2.FlannBasedMatcher()matches = flann.knnMatch(desc1, desc2, k=2)
```

Ratio test (Lowe's method) helps filter good matches by comparing distances between best and second-best matches

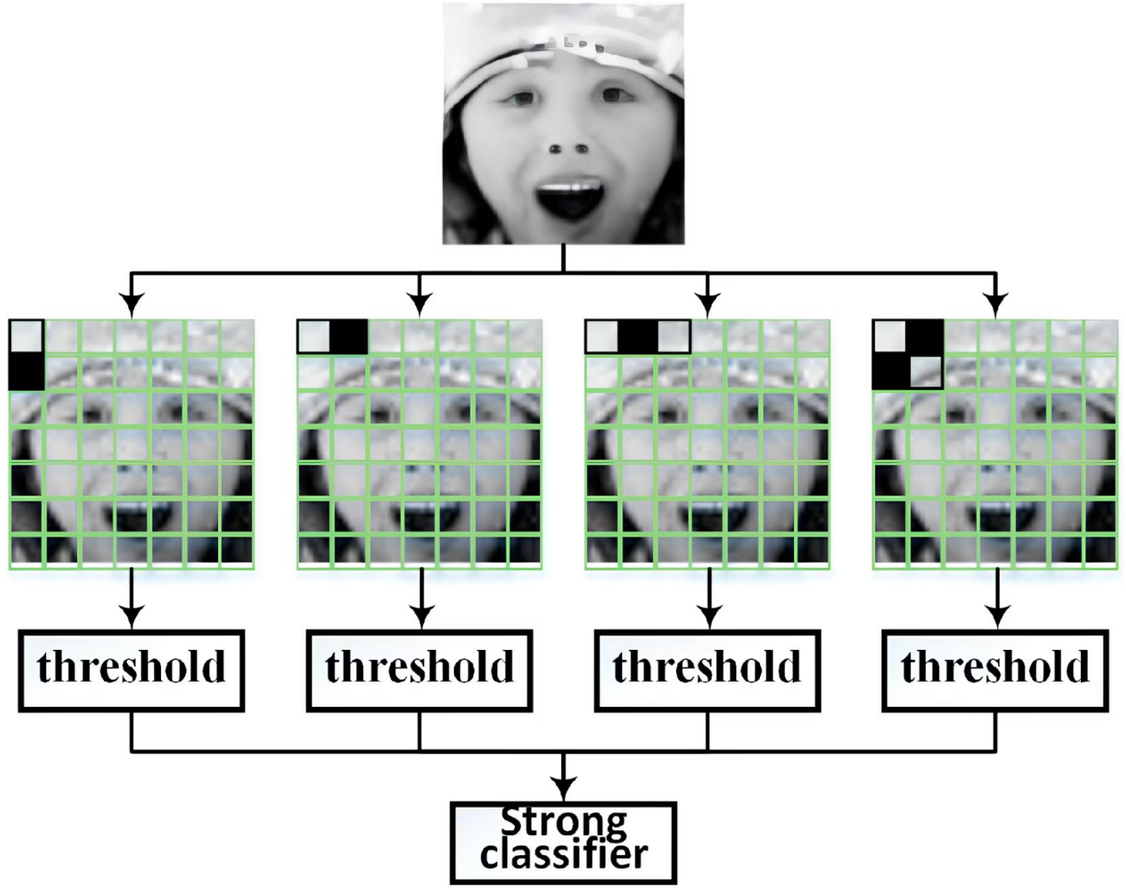
Real-World Application: Face Detection

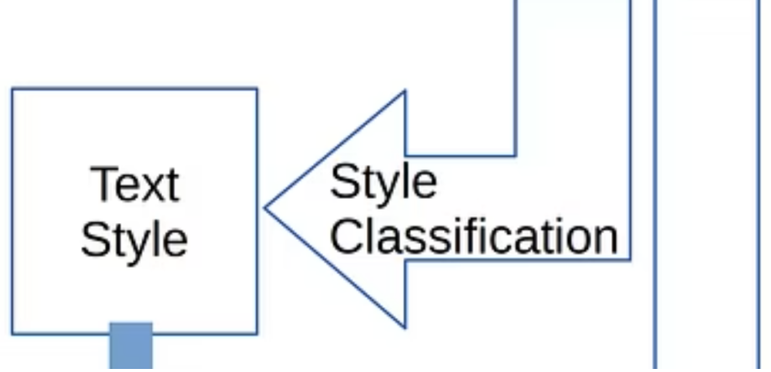
Haar Cascade Classifiers

A popular approach for real-time face detection:

- Uses Haar-like features to detect facial patterns
- Employs cascade of classifiers for efficient detection
- Pre-trained models available in OpenCV
- Works surprisingly well despite being developed in 2001
- Fast enough for real-time applications

```
# Load face cascadeface_cascade = cv2.CascadeClassifier(
'haarcascade_frontalface_default.xml')# Detect facesfaces =
face_cascade.detectMultiScale(    gray, 1.3, 5)    # Draw
rectangles around facesfor (x,y,w,h) in faces:
cv2.rectangle(img, (x,y), (x+w,y+h), (255,0,0), 2)
```





Real-World Application: Optical Character Recognition (OCR)

Converting Images of Text into Machine-Readable Text

Preprocessing

Binarization, deskewing, noise removal, and segmentation to isolate text regions

Character Recognition

Feature extraction and classification to identify individual characters

Post-Processing

Spell checking, grammar correction, and contextual analysis to improve accuracy

Modern OCR systems like Tesseract combine traditional CV techniques with deep learning for improved accuracy across fonts, languages, and image qualities.

```
# Using Tesseract with Pythonimport pytesseracttext = pytesseract.image_to_string(Image.open('text_image.jpg'))
```

Real-World Application: Augmented Reality

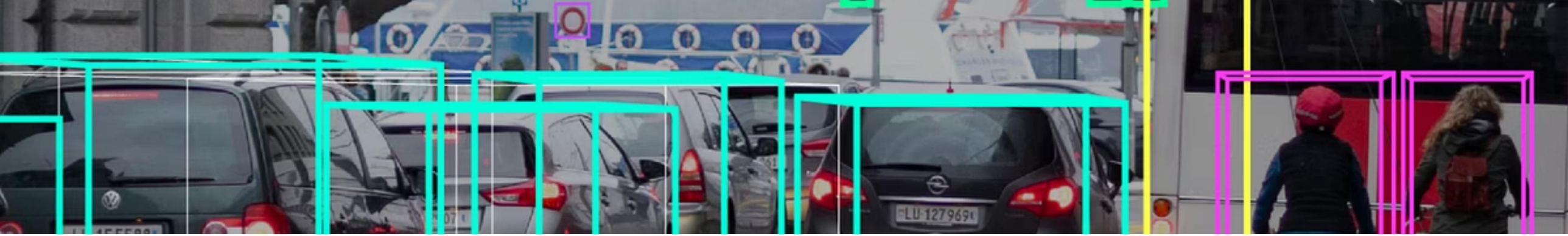
Overlaying Virtual Content on Real World

AR applications rely heavily on computer vision techniques:

- **Feature Detection:** Identify points to track in the environment
- **Pose Estimation:** Determine camera position relative to real world

- **3D Reconstruction:** Build model of surroundings for realistic placement
- **Object Recognition:** Identify real-world objects for interaction
- **SLAM:** Simultaneous Localization And Mapping for persistent AR

Popular frameworks like ARKit (iOS) and ARCore (Android) handle these complex CV tasks through simple APIs, democratizing AR development.



Real-World Application: Object Tracking

Following Objects Through Video Sequences

Traditional Methods

- **Mean-Shift:** Iteratively finds local maxima of density function
- **CAMShift:** Continuously adaptive version of Mean-Shift
- **Optical Flow:** Tracks movement of pixels between frames
- **KCF:** Kernelized Correlation Filters for fast tracking

```
# Lucas-Kanade optical flowp1, st, err =
cv2.calcOpticalFlowPyrLK(    old_gray, frame_gray, p0, None)
```

Modern Approaches

- **SORT/DeepSORT:** Combines detection with Kalman filtering
- **SiamRPN++:** Siamese networks with region proposal
- **GOTURN:** Generic Object Tracking Using Regression Networks
- **FairMOT:** Fair Multi-Object Tracking with unified detection and embedding

Applications include sports analytics, surveillance, autonomous vehicles, and motion capture

Challenges in Computer Vision

Viewpoint Variation

Same object appears different when viewed from different angles

Illumination Conditions

Changes in lighting dramatically affect pixel values

Occlusion

Objects may be partially hidden behind other objects

Scale Variations

Objects appear at different sizes depending on distance

Deformation

Non-rigid objects change shape

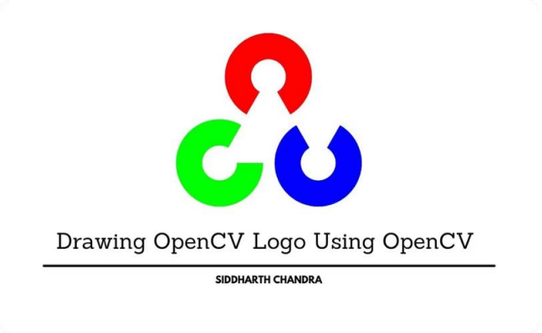
Background Clutter

Objects blend with similarly colored backgrounds

Intra-class Variation

Objects in same category can look very different

Computer Vision Libraries Ecosystem

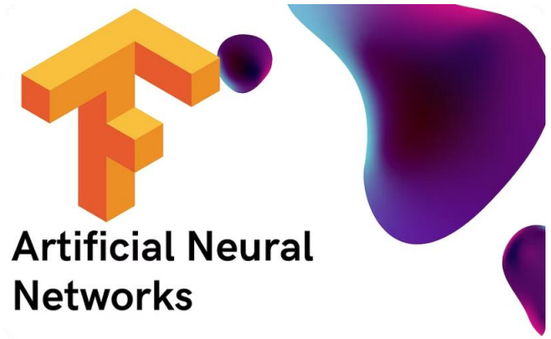


OpenCV

Open source library with 2500+ algorithms for classical CV and ML. Available in C++, Python, Java. Industry standard for traditional CV approaches.

PyTorch

Deep learning library with extensive support for computer vision through torchvision. Preferred in research for dynamic computation graphs.



TensorFlow

Google's ML platform with TF-Vision and Object Detection API. Strong deployment options including mobile and edge devices.

scikit-image

Python library for image processing with focus on algorithm education. Simple API and excellent documentation for beginners.

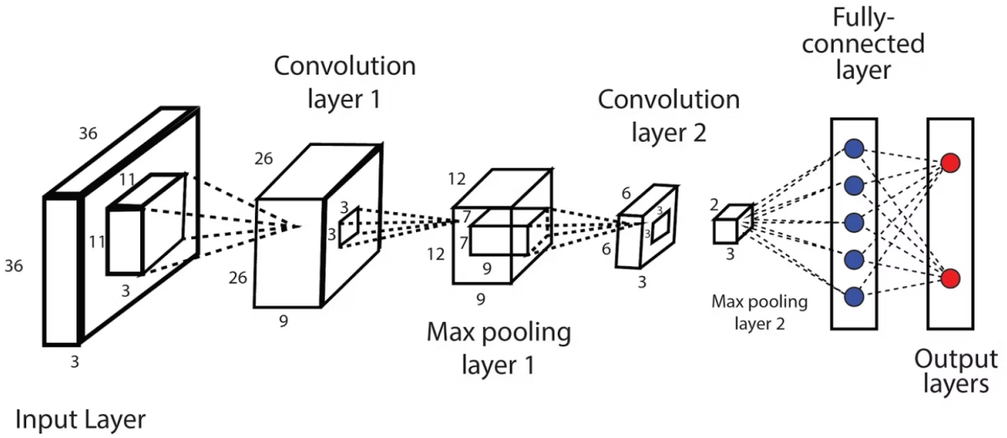
The Rise of Deep Learning in Computer Vision

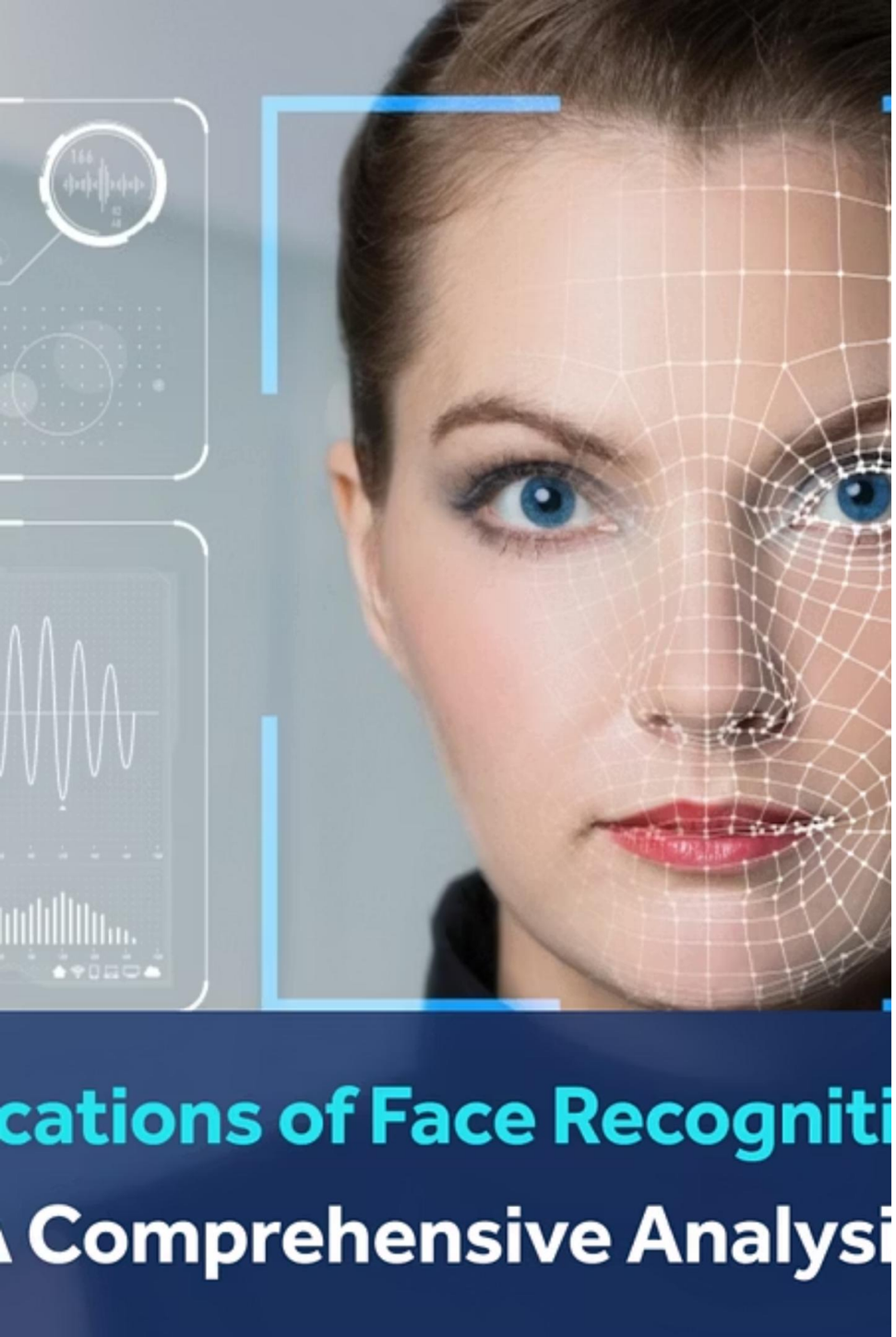
Transforming the Field

Since 2012's AlexNet breakthrough, deep learning has revolutionized computer vision:

- Convolutional Neural Networks (CNNs) automatically learn features from data
- End-to-end training replaces hand-crafted feature engineering
- Performance dramatically outpaces traditional approaches on benchmark tasks
- New architectures like ResNet, Inception, and EfficientNet push accuracy boundaries
- Transfer learning enables application with smaller datasets

While traditional CV techniques remain valuable for many applications, understanding deep learning is increasingly essential for computer vision engineers.





Computer Vision Ethics & Responsible Development

Critical Considerations for CV Engineers

Privacy Concerns

Face recognition, gait analysis, and behavior tracking raise significant privacy issues. Consider data minimization, opt-in consent, and anonymization techniques.

Bias & Fairness

CV systems trained on unrepresentative data show lower accuracy for underrepresented groups. Evaluate performance across demographic groups and mitigate bias through diverse training data.

Transparency

Users should understand when CV systems are in use and how decisions are made. Document limitations and provide explanations for critical applications.

Dual-Use Potential

Many CV technologies have both beneficial and harmful applications. Consider potential misuses during development and implement safeguards where possible.

Hands-On Project Ideas

1

Document Scanner

Build an app that detects document boundaries, applies perspective transform, and enhances readability. Teaches contour detection, perspective transformation, and thresholding.

2

Face Filters

Create Snapchat-like filters by detecting facial landmarks and overlaying graphics. Explores face detection, landmark identification, and image composition.

3

Object Counter

Develop a program that counts objects in images (e.g., cells in microscope images). Covers segmentation, contour analysis, and watershed algorithm.

4

Gesture Control Interface

Build a system that recognizes hand gestures to control applications. Teaches hand detection, contour analysis, and convex hull techniques.

These projects can be completed using OpenCV and Python with relatively little code, making them excellent starting points for building your portfolio.

Future Trends in Computer Vision

Vision Transformers

Moving beyond CNNs, transformer architectures are achieving state-of-the-art results by modeling long-range dependencies in images

Few-Shot Learning

Systems that can learn from very few examples, making CV more accessible for domains with limited labeled data

Multimodal Understanding

Combining vision with language, audio, and other modalities for more holistic scene understanding

Edge AI

Deployment of sophisticated CV models directly on edge devices with privacy preservation and reduced latency

Neuro-symbolic Approaches

Hybrid systems combining neural networks with symbolic reasoning for better interpretability

Self-supervised Learning

Training on unlabeled data by creating clever pretext tasks, reducing dependence on human annotation

Real-World Application: Autonomous Vehicles

Computer Vision as the "Eyes" of Self-Driving Cars

Perception

Multiple cameras, LiDAR, and radar detect objects, lanes, traffic signs, and pedestrians

Scene Understanding

Semantic segmentation classifies each pixel into categories (road, vehicle, pedestrian, etc.)

Prediction

Systems predict future movements of detected objects to anticipate behavior

Planning

Based on visual data, vehicles plan safe trajectories through the environment

CV challenges in autonomous driving include handling adverse weather, recognizing rare events, and ensuring redundancy for safety-critical operations.