Learning Computer Vision from the Pixels Up

Field notes on classical CV, and why I worked through all of it before touching a neural network.

Abir Deol · University of Alberta, Computing Science · July 2026

Abstract. Most people entering computer vision start at a pretrained model and work backwards, which means the first time something breaks they have no mental model to debug against. I took the opposite route: several weeks working through classical CV (convolution, edge detection, segmentation, feature matching, and projective geometry) implemented and tuned by hand in OpenCV before any network was involved. This document is the writeup of that process. It records what each technique actually does, the errors I made and what they taught me, three projects that forced the ideas together, and where the work goes from here. The central claim is simple: a convolutional network is a stack of learned kernels, and it is worth knowing exactly what a kernel does before letting gradient descent choose them for you.

1. Why classical first

The fastest way to produce a working image classifier in 2026 is to load a pretrained backbone, replace the final layer, and fine-tune. It takes an afternoon. It also teaches you nothing about why the model fails on your data, because every operation between input and output is something you imported.

Classical computer vision has the opposite property. Every step is inspectable. When a Canny edge detector produces a broken outline, the failure is traceable to a threshold you chose, a blur you did or did not apply, or noise you never cleaned up. You cannot hide behind the model, because there is no model, only operations you selected and parameters you set. That is a much better environment to be wrong in.

The practical argument is stronger still. A convolutional layer is a bank of learned kernels applied by the same sliding-window convolution used to blur an image by hand. Understanding convolution, receptive fields, and why kernel size must be odd is not preliminary material. It is the same material, arrived at earlier and with the weights chosen by a human instead of an optimizer.

2. The mental model everything rests on

An image is a NumPy array. That is the whole foundation, and holding it firmly makes the rest of the field read as arithmetic rather than magic. A colour image has shape (height, width, channels); grayscale drops to (height, width). The dtype is almost always uint8, one integer from 0 to 255 per channel per pixel.

```
img = cv.imread("image.png")
print(img.shape)   # (height, width, 3)
print(img.dtype)   # uint8
print(img[0, 0])   # top-left pixel as [B, G, R]
```

Once that lands, every operation in the field decomposes. Cropping is array slicing. Colour conversion is a channel reorder or a weighted sum. A blur is a weighted average over a neighbourhood. Nothing in the

pipeline is doing anything a sufficiently patient person could not do by hand with the numbers in front of them.

It also explains why bounding boxes are trivial once detection gives you four numbers. An object detector returns x1, y1, x2, y2, and extraction is one line of slicing: `img[y1:y2, x1:x2]`. The output of a model I had not yet learned to build was already a shape I understood.

3. Convolution, and what a kernel actually is

A kernel is a small array, usually 3x3, that slides across every pixel. At each position it multiplies its values against the surrounding pixels and sums them into one new value. That operation is convolution, and it is the single most reused idea in the discipline.

```
sharpen = np.array([[ 0, -1,  0],
                    [-1,  5, -1],
                    [ 0, -1,  0]])
sharpened = cv.filter2D(img, -1, sharpen)
```

Reading that kernel tells you what it does before you run it. The centre pixel is amplified fivefold while its four neighbours are subtracted, exaggerating local differences, which is what sharpening is. The general rule follows directly: when a kernel's values sum to zero, uniform regions collapse to black and only differences survive. That is the entire basis of edge detection, visible from the array.

Kernel size must be odd, because the kernel needs a single centre pixel to sit on. A 4x4 grid has no centre. This is a small constraint that stops being an arbitrary rule the moment you picture the window sliding.

4. Edges: Canny as a four-stage pipeline

An edge is where brightness changes rapidly: an object boundary, a surface border, a shadow line. The underlying measurement is the gradient, computed with Sobel operators. Canny wraps that measurement in four stages: Gaussian blur to suppress noise, gradient calculation for edge strength and direction, non-maximum suppression to thin edges to single-pixel lines, and hysteresis thresholding.

Hysteresis is the part worth internalizing

- Above the high threshold: definitely an edge, kept unconditionally.
- Below the low threshold: definitely not an edge, discarded.
- Between the two: a weak edge, kept *only if connected to a strong edge*.

That third rule is the design. It is what keeps a real edge continuous through a dim patch while rejecting an isolated bright speck of noise. It is a connectivity argument, not a brightness one. A useful starting ratio is high about two to three times low, e.g. 50 and 150.

I tuned these with trackbars rather than by guessing, which turned a parameter I was choosing arbitrarily into one I had watched behave across its whole range. The structural detail that matters: create the trackbar once, outside the loop; read its position every frame, inside the loop.

```
cv.namedWindow("canny")
```

```
cv.createTrackbar("low",  "canny",  50, 300, lambda x: None)
cv.createTrackbar("high", "canny", 150, 300, lambda x: None)
while True:
    low  = cv.getTrackbarPos("low",  "canny")
    high = cv.getTrackbarPos("high", "canny")
    cv.imshow("canny", cv.Canny(img, low, high))
```

5. Segmentation: thresholding, Otsu, and contours

Thresholding reduces a grayscale image from 256 possible values to 2. That sounds lossy because it is, and it is exactly what shape analysis needs, because most contour and morphology operations require a binary input.

A manual cutoff breaks the moment lighting changes. Otsu's method removes the guess by analysing the histogram and selecting the cutoff that best separates foreground from background. You pass 0 as a placeholder and let it decide.

```
otsu_val, thresh = cv.threshold(
    gray, 0, 255, cv.THRESH_BINARY + cv.THRESH_OTSU)
```

A contour is a curve tracing the boundary of a connected white region, stored as an array of (x, y) points. Because contours carry area, they can be sorted and filtered, and that one fact does most of the practical work. Small contours are almost always noise; real objects have meaningful area. Sorting by `cv.contourArea` and keeping the largest few is the backbone of both projects in section 7.

When thresholding leaves a speckled mess, morphological operations clean it: erosion shrinks white regions and removes specks, dilation grows them and fills gaps, opening removes noise while preserving object size, and closing fills holes inside objects. These are not a separate topic so much as the repair kit that makes the previous step usable.

6. Features: describing a point so it can be found again

A feature is a distinctive, repeatable point: a corner or blob recognizable from a different viewpoint. Flat regions make poor features; corners make excellent ones. The distinction that took me longest to hold cleanly is that feature work is two separate steps, not one:

- **Keypoint**: the location, size, and orientation. Answers *where?*
- **Descriptor**: a numeric fingerprint of the surrounding patch. Answers *what does it look like?*

ORB combines the FAST corner detector with the BRIEF binary descriptor and adds rotation handling. It is fast, unencumbered, and good enough for most work. SIFT produces more robust matches across scale and rotation using floating-point descriptors, at a real speed cost. ORB is the default; SIFT is what you reach for when accuracy is the constraint rather than latency.

Matching compares descriptors by distance, and the distance metric must match the descriptor type: Hamming for ORB's binary strings, L2 for SIFT's floating-point vectors. Getting this wrong produces matches that are not errors so much as nonsense, which is a harder failure to notice.

```
orb = cv.ORB_create()
kp1, des1 = orb.detectAndCompute(img1, None)
kp2, des2 = orb.detectAndCompute(img2, None)

bf = cv.BFMatcher(cv.NORM_HAMMING, crossCheck=True)
matches = sorted(bf.match(des1, des2), key=lambda m: m.distance)
```

`crossCheck=True` only accepts a match when both descriptors independently select each other, which removes a large share of false positives for one keyword. The visual sanity check is geometric: correct match lines run roughly parallel, and chaotic crossing lines mean the matches are wrong regardless of what the distances say.

7. Geometry: homography

A homography is a 3x3 matrix describing how one flat plane maps to another flat plane viewed from a different angle. It is what lets a photograph taken at an angle be warped to a head-on view, and it is the point where feature matching stops being a demo and becomes infrastructure.

```
M = cv.getPerspectiveTransform(src_pts, dst_pts)
warped = cv.warpPerspective(img, M, (out_w, out_h))
```

With exactly four corresponding corners you can solve it directly. With many noisy feature matches instead, `cv.findHomography(src, dst, cv.RANSAC)` estimates it robustly by repeatedly fitting to random subsets and keeping the consensus, which is how panorama stitching aligns overlapping photos.

8. Three projects that forced it together

Live webcam colour filter

Captures video, converts each frame to HSV, builds a binary mask for one colour range, and keeps only those pixels. HSV matters here because separating hue from brightness makes colour selection robust to lighting in a way RGB never is. `cv.inRange` produces the mask; `cv.bitwise_and` zeroes everything outside it. Small and immediate, and the first time capture, colour space, masking, and bitwise operations all had to work at once.

Document scanner

Photograph a document at an angle and flatten it to a top-down crop: grayscale and blur, Canny for edges, find contours and sort by area, approximate the largest to a polygon, and take the one that reduces to four corners. Those four points become the source for a perspective transform. This is the project where the pieces stopped being separate techniques. Edges feed contours, contours feed geometry, geometry produces the output.

```
for c in sorted(cnts, key=cv.contourArea, reverse=True):
    peri   = cv.arcLength(c, True)
    approx = cv.approxPolyDP(c, 0.02 * peri, True)
    if len(approx) == 4:      # a document-shaped quadrilateral
        doc = approx
        break
```

Motion detector

Compare consecutive frames; where pixels changed, something moved. `cv.absdiff` gives the per-pixel difference, thresholding turns it into a binary motion mask, dilation fills gaps, and an area filter discards small contours so only real movement gets a bounding box. It is crude compared to a tracker, and it is also the honest primitive underneath one.

9. What I got wrong

These are the errors that cost me actual hours. Every one is obvious in hindsight and none was obvious at the time, which is the useful property of a mistake.

- **BGR vs RGB.** OpenCV loads BGR; Matplotlib expects RGB. Skip the conversion and reds and blues silently swap. The image looks plausible, just wrong.
- **Colormaps.** A 2D single-channel array needs `cmap='gray'`. A 3D array does not. Checking `img.shape` first answers this every time.
- **Printing an array instead of the image.** Printing `img` shows raw pixel numbers. That is correct behaviour, not a bug. Rendering is a separate step.
- **imread returning None.** Almost always a wrong path, not a corrupt file. Check the working directory before anything else.
- **Mismatched distance metrics.** Hamming belongs to ORB, L2 to SIFT. Mixing them produces confidently meaningless matches.
- **Live loops in notebooks.** Webcam and trackbar loops need a real script. This is an environment constraint, not a code error, and it is invisible until you hit it.
- **Mutating the original.** Draw on `img.copy()`. Otherwise every experiment starts from the residue of the last one.

10. Where this goes next

The roadmap from here moves into learned representations, and the intent is to carry the same discipline forward: build the primitive before importing the abstraction.

- **Tensors and preprocessing.** Resizing, normalization, and augmentation, already array work, now shaped (batch, channels, height, width). The 224x224 input size that recurs across pretrained backbones is a convention worth knowing by reflex.
- **Convolutional networks.** The direct continuation of section 3: the same sliding window, with kernel values learned rather than chosen. Understanding what a hand-written sharpening kernel does is what makes a first-layer filter visualization legible.
- **Detection and tracking.** Where the motion detector gets replaced by something that holds identity across frames, and where bounding boxes stop being slicing exercises.
- **Building rather than fine-tuning.** The end state is a project that is mine rather than a tutorial's, in the same spirit as writing an allocator instead of calling malloc.

The reason for the ordering is the through-line of this whole document. Every shortcut available in this field trades understanding for speed, and the trade is usually worth it, but only once, and only after you know what you gave up.

Tooling: Python, OpenCV, NumPy, Matplotlib. Notes and project source at github.com/apollo-2006. This document is a working record and will be revised as the roadmap progresses.