**Training Notebooks for Beginner Computer Vision Tasks**

To get our student up to speed, we will create **four Jupyter notebooks**, each focusing on one fundamental computer vision task: **(1)** Image Classification, **(2)** Image Segmentation, **(3)** Video Object Tracking, and **(4)** Video-Based Movement Analysis. Each notebook will use **open-source, free datasets**, include **accessible code examples** (with heavy use of **OpenCV** for image processing), and provide clear markdown explanations with links to documentation or relevant code repositories. The emphasis is on simplicity and clarity, ensuring the student can follow along and experiment.

Below, we outline each notebook’s content and structure. For each, we list the dataset, the approach or model used, and the step-by-step flow with example code snippets and explanations. Markdown cells will guide the student through the process, and code cells will demonstrate the concepts in action.

A white background with black text

AI-generated content may be incorrect.

**Notebook 1: Image-Based Classification**

**Goal:** Teach the student how to classify images into categories using a simple model. We’ll use the **MNIST** dataset, a beginner-friendly dataset of handwritten digits. This notebook will show how to load data, visualize it, train a model, and evaluate performance. We’ll integrate OpenCV by using it to manipulate and display images.

**Dataset:** *MNIST* is a classic dataset with 60,000 training images and 10,000 test images of handwritten digits (0 through 9) in 28×28 pixel grayscale format. It's built into many libraries, making it trivial to load. In the notebook, a markdown cell will note: *“We use the MNIST dataset. Each image is 28x28 pixels of a handwritten digit, and there are 10 classes (digit 0-9).”* We will also provide a link to the [Keras documentation for MNIST](https://keras.io/api/datasets/mnist/) for reference.

**Step 1 - Loading the Dataset:**\ A code cell will demonstrate loading MNIST using a library (to avoid manual downloading). For example, with TensorFlow/Keras:

from tensorflow.keras.datasets import mnist

# Load data, splitting into train and test sets

(X\_train, y\_train), (X\_test, y\_test) = mnist.load\_data()

print(f"Train: {X\_train.shape}, Train labels: {y\_train.shape}")

print(f"Test: {X\_test.shape}, Test labels: {y\_test.shape}")

*Markdown explanation:* This cell uses Keras to fetch MNIST. The student learns that X\_train is a NumPy array of shape (60000, 28, 28) and y\_train contains the labels. We’ll mention that this is a convenient way to get standard datasets. (Alternatively, we could link to how to load via OpenCV if images were local, e.g. using cv2.imread in a loop, but built-in loader is simpler.) If Keras/TensorFlow is not available, we can mention alternatives like using sklearn.datasets.fetch\_openml('mnist\_784') or a direct download.

**Step 2 - Visualizing Sample Images:**\ We include a code cell to visualize some digits so the student sees what the data looks like. We can use Matplotlib for inline display, or OpenCV’s imshow if running locally (though in notebooks Matplotlib is easier). For instance:

import matplotlib.pyplot as plt

plt.figure(figsize=(5,5))

for i in range(9):

plt.subplot(3,3,i+1)

plt.imshow(X\_train[i], cmap='gray')

plt.title(f"Label: {y\_train[i]}")

plt.axis('off')

plt.show()

¨G0G

The student should see ~0.97 (97%) accuracy in a couple of minutes of training. We’ll mention that this is much better than the k-NN classifier, illustrating the power of CNNs for vision.

* We will clarify that *training the CNN requires a machine with some processing power or a GPU*, but since the model is small and dataset grayscale, CPU training in a reasonable time is possible (~ a few minutes). If performance is an issue, we could reduce epochs or sample size.

**Step 5 - Evaluation and Visualization of Results:**\ Finally, we add some code to evaluate and visualize results:

* For the CNN: display a **confusion matrix** or a few test images with their predicted labels vs true labels to see where the model succeeds or fails.
* import numpy as np
* preds = model.predict(X\_test\_exp)
* pred\_labels = preds.argmax(axis=1)
* # Pick 5 random test images and show predictions
* idxs = np.random.choice(len(X\_test), 5, replace=False)
* for i in idxs:
* plt.imshow(X\_test[i], cmap='gray')
* plt.title(f"True: {y\_test[i]}, Predicted: {pred\_labels[i]}")
* plt.show()

*Markdown:* Note how many of these random samples the model got right. If a mistake occurs, discuss possible reasons (maybe a weird handwriting style). This helps the student learn to **interpret model output**.

* For k-NN: maybe just report overall accuracy or show one or two classification results similarly. The CNN coverage is more important, so we won’t dwell too long on k-NN beyond introducing it.

**Links to Docs/Repos:** Throughout the markdown, we will link relevant resources:

* “For more on k-NN classification with OpenCV, see the official tutorial[[1][1]](https://stackoverflow.com/questions/9413216/simple-digit-recognition-ocr-in-opencv-python).”
* “Keras provides more <https://github.com/keras-team/keras-io/blob/master/examples/vision/mnist_convnet.py> in their GitHub repository.”
* We might link to a well-known blog like the GfG article on MNIST with Keras/PyTorch for further reading (as a beginner might find it useful).
* We also ensure all datasets and libraries used are free: Keras/TensorFlow and sklearn are all open-source; MNIST is public domain (or effectively free to use).

By the end of Notebook 1, the student will have:

* **Hands-on experience** loading and displaying image data.
* Built two simple models and seen their performance.
* An understanding of how OpenCV can be used alongside model training (for visualization or augmentation).
* Exposure to documentation and further resources to deepen their understanding.

A white background with black text

AI-generated content may be incorrect.

**Notebook 2: Image-Based Segmentation**

**Goal:** Introduce segmenting an image into regions (foreground/background or by object classes). This notebook focuses on **semantic segmentation** at a basic level (labeling each pixel), using classical (non-neural) methods for simplicity. The student will learn how to create binary masks and use OpenCV algorithms like thresholding, contour detection, watershed, and GrabCut. We will also mention how these relate to more advanced deep learning methods and provide links to those, but the hands-on will be with easier techniques.

**Data:** Instead of a big dataset, we’ll use a **single representative image** (or a few) because segmentation ground truth can be complex. A great example is the classic **“Messi” image** often used in OpenCV docs (Lionel Messi playing football)[[2]](https://docs.opencv.org/3.4/d8/d83/tutorial_py_grabcut.html). In that image, we can try to segment Messi (the player) from the background. Alternatively, we could use an image of coins on a plain background for watershed segmentation, or the Oxford-IIIT Pet dataset (which has masks for animals). To keep it simple:

* We’ll include one image file in the environment (e.g., messi5.jpg, which is in OpenCV’s sample images).
* Also possibly a second example: an image with **two touching objects**, like coins or overlapping objects, to illustrate watershed segmentation.

**Step 1 - Load and Display Image:**\ Use OpenCV to read the image:

import cv2

import numpy as np

import matplotlib.pyplot as plt

img = cv2.imread('messi5.jpg')

# OpenCV loads in BGR, convert to RGB for displaying in matplotlib

img\_rgb = cv2.cvtColor(img, cv2.COLOR\_BGR2RGB)

plt.imshow(img\_rgb)

plt.title("Original Image")

plt.axis('off')

plt.show()

*Markdown:* Explain that OpenCV’s imread loads the image as a NumPy array. We note the shape (e.g., (342, 548, 3) for height, width, channels). We convert BGR to RGB for correct color display in Matplotlib. If the student runs in a local environment, they could also use cv2.imshow to pop up a window (with cv2.waitKey), but in a notebook, Matplotlib is more convenient.

**Step 2 - Basic Thresholding (for simple segmentation):**\ We’ll first demonstrate converting the image to grayscale and doing global thresholding to create a binary mask. This shows the simplest form of segmentation (separating foreground/background by intensity).

gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

# Otsu's threshold to automatically find a good cutoff

\_, mask = cv2.threshold(gray, 0, 255, cv2.THRESH\_BINARY\_INV + cv2.THRESH\_OTSU)

plt.imshow(mask, cmap='gray')

plt.title("Binary mask (Otsu threshold)")

plt.axis('off')

plt.show()

*Markdown:* We explain Otsu’s method briefly (automatic threshold calculation) and why we used THRESH\_BINARY\_INV (maybe the object was brighter than background so we invert to get object white). We clarify that mask is a binary image where white pixels (255) correspond to the segmented foreground (e.g., the player) and black (0) to background. Depending on the image, thresholding might not perfectly isolate the object (e.g., if background has similar intensity). This is a segue into more sophisticated methods.

**Step 3 - Contour Detection:**\ Using the binary mask, find contours which hopefully outline the object of interest.

contours, \_ = cv2.findContours(mask, cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE)

# Draw the contour on a copy of the image

contour\_img = img\_rgb.copy()

cv2.drawContours(contour\_img, contours, -1, (0,255,0), 2)

plt.imshow(contour\_img)

plt.title("Contours found")

plt.axis('off')

plt.show()

*Markdown:* We tell the student that findContours retrieves the boundaries of connected components in the mask[[1]](https://stackoverflow.com/questions/9413216/simple-digit-recognition-ocr-in-opencv-python). Here we used RETR\_EXTERNAL to get external contours (assuming one main object). We draw the contour in green. The output image shows the contour around Messi (for example). We note that contour detection is useful for shape analysis or counting objects after segmentation.

If the threshold step didn’t perfectly isolate the object, we might see extra contours or missing parts. This leads to discussing improvements.

**Step 4 - Watershed for Separating Touching Objects (optional):**\ If we have a second example (like coins or two overlapping objects), we can illustrate the watershed algorithm to separate them. This is optional but can be very insightful.

* Use a provided image coins.png (or generate one) with overlapping coins.
* Convert to grayscale, apply threshold.
* Use cv2.distanceTransform and markers to apply cv2.watershed.

Example snippet:

# Assuming 'coins.jpg' is an image with touching coins

coins = cv2.imread('coins.jpg')

gray\_coins = cv2.cvtColor(coins, cv2.COLOR\_BGR2GRAY)

\_, thresh = cv2.threshold(gray\_coins, 0, 255, cv2.THRESH\_BINARY+cv2.THRESH\_OTSU)

# Noise removal via morphology

kernel = np.ones((3,3), np.uint8)

opening = cv2.morphologyEx(thresh, cv2.MORPH\_OPEN, kernel, iterations=2)

# Sure background area

sure\_bg = cv2.dilate(opening, kernel, iterations=3)

# Sure foreground area via distance transform

dist = cv2.distanceTransform(opening, cv2.DIST\_L2, 5)

\_, sure\_fg = cv2.threshold(dist, 0.7\*dist.max(), 255, 0)

sure\_fg = np.uint8(sure\_fg)

# Unknown region (border of objects)

unknown = cv2.subtract(sure\_bg, sure\_fg)

# Marker labelling

\_, markers = cv2.connectedComponents(sure\_fg)

markers = markers + 1

markers[unknown==255] = 0

markers = cv2.watershed(coins, markers)

coins\_marked = coins.copy()

coins\_marked[markers == -1] = [255,0,0] # mark boundaries in red

plt.imshow(cv2.cvtColor(coins\_marked, cv2.COLOR\_BGR2RGB))

plt.title("Watershed segmentation")

plt.axis('off'); plt.show()

*Markdown:* This code is a bit advanced. We will break it down in the markdown: noise removal with morphological opening, computing **distance transform** to separate objects, and then using watershed which treats the distance map peaks as seed points (markers). The student is not expected to grasp every line, but we want to expose them to a canonical segmentation technique. We reference the OpenCV Watershed tutorial for in-depth explanation, encouraging them to read it if interested. The result should show each coin separated with red boundaries between them, demonstrating how watershed resolves overlapping objects.

**Step 5 - GrabCut for Foreground Extraction:**\ Now we demonstrate a user-assisted segmentation using the GrabCut algorithm, which is great for extracting a complex foreground from background with minimal input[[2]](https://docs.opencv.org/3.4/d8/d83/tutorial_py_grabcut.html). We use our main image (Messi) again:

mask = np.zeros(img.shape[:2], np.uint8)

bgdModel = np.zeros((1,65), np.float64)

fgdModel = np.zeros((1,65), np.float64)

# Define a rectangle around the object (x, y, w, h)

rect = (50, 50, 450, 290) # these coords should tightly encompass the player

cv2.grabCut(img, mask, rect, bgdModel, fgdModel, 5, cv2.GC\_INIT\_WITH\_RECT)

# Process the mask to binary segmentation

mask\_final = np.where((mask==cv2.GC\_FGD) | (mask==cv2.GC\_PR\_FGD), 1, 0).astype('uint8')

# Apply mask to image

segmented = img \* mask\_final[:,:,np.newaxis]

plt.imshow(cv2.cvtColor(segmented, cv2.COLOR\_BGR2RGB))

plt.title("GrabCut Output")

plt.axis('off'); plt.show()

*Markdown:* We explain the parameters: we initialized a mask and background/foreground models (temporary arrays for the algorithm). We provided a rectangle that bounds Messi[[2]](https://docs.opencv.org/3.4/d8/d83/tutorial_py_grabcut.html) – this is the only user input needed. After grabCut, the mask is filled with codes: 0 = background, 1 = foreground, 2 = probable background, 3 = probable foreground[[2]](https://docs.opencv.org/3.4/d8/d83/tutorial_py_grabcut.html). The line with np.where converts this mask to a binary form (1 for sure or probable foreground, 0 for rest)[[2]](https://docs.opencv.org/3.4/d8/d83/tutorial_py_grabcut.html). We multiply this mask with the original image to get the segmented foreground alone[[2]](https://docs.opencv.org/3.4/d8/d83/tutorial_py_grabcut.html). The output image shows Messi cut out from the background. The markdown will have something like: *“GrabCut iteratively refines the segmentation. We gave an initial rectangle, and after 5 iterations, it produced this result. Notice the background is now blacked out and only the foreground remains*[*[2]*](https://docs.opencv.org/3.4/d8/d83/tutorial_py_grabcut.html)*.”* If the result misses some parts (like the example in docs misses some hair[[2]](https://docs.opencv.org/3.4/d8/d83/tutorial_py_grabcut.html)), we mention that GrabCut can be improved with additional user strokes, but we won’t implement the interactive refinement here. We’ll link to the OpenCV GrabCut tutorial for details[[2]](https://docs.opencv.org/3.4/d8/d83/tutorial_py_grabcut.html).

**Step 6 - Visualizing and Saving Masks:**\ We show how to overlay the segmentation on the original image for presentation:

# Create an overlay: original image with mask boundary

overlay = img\_rgb.copy()

overlay[mask\_final==0] = [0,0,0] # black out background in RGB image

plt.imshow(overlay); plt.title("Segmented Foreground (overlay)"); plt.axis('off'); plt.show()

Also, demonstrate saving the mask or result:

cv2.imwrite('messi\_mask.png', mask\_final\*255) # save the mask as an image

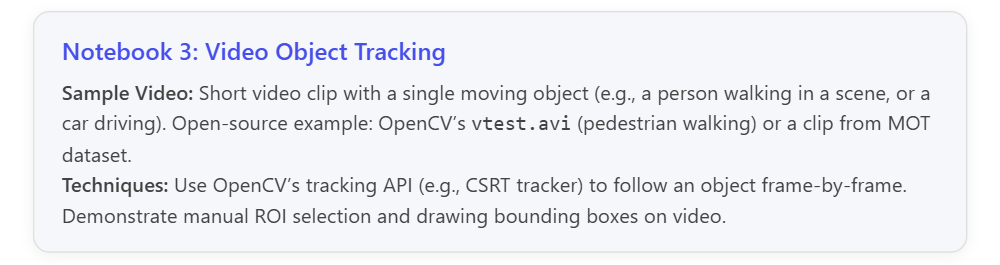
*Markdown:* Emphasize that saving results is important for later use (like feeding masks into other pipelines or manually checking them).

**Discussion and Links:**\ To conclude the notebook, we’ll have a markdown section discussing how what we did relates to more advanced segmentation:

* *“In practice, modern image segmentation often uses deep learning (e.g., U-Net, Mask R-CNN) to achieve more accurate results. For instance, the Pascal VOC dataset and MS COCO provide ground-truth masks for many classes, and models like Mask R-CNN can predict those. While we won’t train a deep model here, you can try using a pre-trained Mask R-CNN on an image via OpenCV’s DNN or a library like Detectron2.”* We provide a link to a GitHub or tutorial for using Mask R-CNN (e.g., the PyImageSearch Mask R-CNN tutorial or matterport’s Mask R-CNN implementation).
* Link the OpenCV documentation for segmentation functions: threshold, findContours, watershed, grabCut[[2]](https://docs.opencv.org/3.4/d8/d83/tutorial_py_grabcut.html) so the student can read more details.
* Possibly reference the **scikit-image library** as another resource that has many segmentation algorithms (for instance, scikit-image’s slic for superpixel segmentation). We might say: *“Another approach is superpixels (e.g., SLIC algorithm) – OpenCV has cv2.ximgproc.createSuperpixelSLIC if you want to explore that.”* with a link to the OpenCV Extended docs.

By the end of Notebook 2, the student will:

* Understand how to create binary masks to separate foreground/background.
* Know how to use OpenCV for thresholding and finding contours.
* Have seen an advanced algorithm (GrabCut) in action for segmentation with minimal input.
* Be aware of how classical methods compare to deep learning segmentation (and have pointers to learn more).



**Notebook 3: Video Object Tracking**

**Goal:** Teach how to track an object across video frames, preserving its identity. The notebook will use OpenCV’s built-in **single-object trackers** to track a moving object in a video. The student will learn how to work with video streams frame-by-frame, initialize a tracker with an ROI, update it, and visualize the tracking with bounding boxes.

**Data:** We need a video with a discernible object to track. Options:

* OpenCV includes a test video vtest.avi (a surveillance camera clip of a person walking) which is commonly used in OpenCV tutorials for background subtraction. We can use that or any short video (10-20 seconds) of one object moving.
* Alternatively, the **OTB dataset** or **MOT Challenge** sequences have videos for tracking tasks. To keep it simple, we can supply one video file (e.g., face.mp4 of a person’s face moving, or car.mp4).
* If no video file is available, we can mention using a webcam feed via OpenCV (cv2.VideoCapture(0)) to test tracking in real-time on whatever the camera sees.

Assume we have object.mp4 in the working directory for this training.

**Step 1 - Load the Video:**\ Introduce OpenCV’s VideoCapture:

cap = cv2.VideoCapture('object.mp4')

if not cap.isOpened():

print("Error: Could not open video")

else:

fps = cap.get(cv2.CAP\_PROP\_FPS)

frame\_count = int(cap.get(cv2.CAP\_PROP\_FRAME\_COUNT))

print(f"Video opened. Frame count: {frame\_count}, FPS: {fps:.2f}")

*Markdown:* Explain that VideoCapture can open a video file or camera. We check isOpened to ensure it succeeded. We also retrieved FPS and frame count for information. We’ll mention that OpenCV can retrieve frames one by one using cap.read().

**Step 2 - Read First Frame & Select ROI:**\ We need an initial bounding box of the object to track. We can do this manually or using a UI. Easiest is to use cv2.selectROI which lets user draw a rectangle on the frame:

ret, frame = cap.read()

frame\_display = frame.copy()

if not ret:

print("Can't read the video file.")

else:

# Show frame and use selectROI for user to draw box

bbox = cv2.selectROI("Frame", frame\_display, False)

cv2.destroyWindow("Frame")

print("Selected ROI:", bbox)

At this point, the user will manually draw a rectangle around the object in the displayed frame (when running interactively). If this is not feasible (e.g., if running non-interactively), we can alternatively predefine bbox = (x, y, w, h) from known ground truth just for demonstration. But we’ll encourage the student to try the interactive mode, as it’s a useful skill to manually initialize trackers.

*Markdown:* Explain that cv2.selectROI is a convenience function that opens a window where you can drag a bounding box and press Enter or Space to confirm[[3]](https://woteq.com/object-tracking-with-csrt-tracker-in-python-using-opencv/). We note that the returned bbox is a tuple (x, y, width, height). If the video has a clear subject, the student should draw around it (e.g., around the walking person or moving car).

**Step 3 - Initialize Tracker:**\ OpenCV offers multiple tracking algorithms (we’ll mention CSRT, KCF, MIL, etc.). We’ll use **CSRT** as it is accurate for beginners and doesn’t require additional model files. Initialize the tracker with the first frame and ROI:

tracker = cv2.TrackerCSRT\_create()

ok = tracker.init(frame, bbox)

if ok:

print("Tracker initialized successfully")

else:

print("Tracker initialization failed")

*Markdown:* We explain that TrackerCSRT\_create() gives us a tracker object using the CSRT algorithm[[3]](https://woteq.com/object-tracking-with-csrt-tracker-in-python-using-opencv/). We then call init(image, bbox) with the first frame and the selected region. If ok is True, the tracker is ready. We might link to OpenCV’s tracking API documentation, or mention alternatives: *“OpenCV also has TrackerKCF (faster but less accurate), MIL, TLD, etc., which can be created similarly. CSRT is chosen for its robustness*[*[3]*](https://woteq.com/object-tracking-with-csrt-tracker-in-python-using-opencv/)*.”*

**Step 4 - Tracking Loop:**\ Now we loop over the remaining frames to update the tracker and draw the bounding box:

frame\_idx = 1

while True:

ret, frame = cap.read()

if not ret:

break

frame\_idx += 1

ok, bbox = tracker.update(frame)

if ok:

# Tracking success: draw bbox

x, y, w, h = map(int, bbox)

cv2.rectangle(frame, (x, y), (x+w, y+h), (0,255,0), 2)

cv2.putText(frame, "Tracking", (10,30), cv2.FONT\_HERSHEY\_SIMPLEX,

1, (0,255,0), 2)

else:

# Tracking failure

cv2.putText(frame, "Lost track", (10,30), cv2.FONT\_HERSHEY\_SIMPLEX,

1, (0,0,255), 2)

# Display frame (in a notebook, we might skip real-time display and just save a few frames)

cv2.imshow("Tracking", frame)

if cv2.waitKey(30) &amp; 0xFF == 27: # ESC key to exit

break

cap.release()

cv2.destroyAllWindows()

*Markdown:* Explain step-by-step:

* tracker.update(frame) attempt to locate the object in the new frame[[3]](https://woteq.com/object-tracking-with-csrt-tracker-in-python-using-opencv/). It returns the updated bbox and a flag.
* If ok is True, we draw a rectangle on the frame at the new position. We use cv2.rectangle with green color. We also put a text "Tracking" in green to indicate success. If ok is False (tracker lost the object), we display "Lost track" in red.
* cv2.imshow shows the frame. In a Jupyter environment, imshow might not work inline. So, if running in Jupyter, we have two options:
  1. Use cv2.waitKey to create a window as above (works if the environment supports it).
  2. Or collect frames and display them with matplotlib periodically. We could store the frames with bounding boxes in a list and later show a few snapshots or even create a small animation using matplotlib. For simplicity, instruct the student that if cv2.imshow doesn’t display (e.g., in a Jupyter hosted environment like Colab), they can add plt.imshow(cv2.cvtColor(frame, cv2.COLOR\_BGR2RGB)) inside the loop with a short pause, or they can run the notebook locally.
* We use waitKey(30) to delay ~30 ms between frames (assuming ~30 FPS video), and allow ESC to break out.
* At the end, release the video and destroy windows (good practice to free resources).

This will produce a video playback with the rectangle moving following the object. In the markdown, we’ll describe what the student should observe: e.g., *“The green box follows the person walking across the frame. If the person exits or gets occluded, the tracker might lose them and display 'Lost track'. Try tracking different objects or videos. For example, if you track a car in a driving video, see how well it stays locked on.”*

We’ll also mention measuring performance: *“We are drawing at ~30 FPS; we can compute actual FPS by timing the loop if needed. The code prints 'Tracking' and the bounding box coordinates each frame. For more efficient visualization in a notebook, consider saving the output frames and creating a video file or GIF.”*

**Step 5 - Experiment with Different Trackers (optional):**\ We can allow the student to try other trackers easily:

# Switch to a different tracker, e.g., KCF

tracker = cv2.TrackerKCF\_create()

We’ll note in markdown: *“You can replace TrackerCSRT\_create() with TrackerKCF\_create() or others. KCF might be faster but can fail on occlusion. You could also track multiple objects by creating a MultiTracker and adding multiple ROIs, see OpenCV's docs on MultiTracker.”* We link to sources or tutorials (like LearnOpenCV’s multi-object tracking guide which shows how to add several ROIs to a MultiTracker).

**Bonus - Using Pre-trained Detector + Tracking:**\ If time permits in the material, we mention: *“Often tracking is combined with detecting the object in each frame. For instance, OpenCV’s DNN module can detect objects (like people or cars using YOLO or SSD models), and then a tracker can keep following the detected object between detection frames for efficiency. A simple approach is the cv2.legacy.TrackerCSRT we used. For multi-object tracking in crowded scenes, algorithms like SORT or DeepSORT (open-source on GitHub) use detection+Kalman filter+Hungarian algorithm to match objects across frames.”* We keep this high-level, but provide pointers: maybe link to the DeepSORT GitHub or an article.

**Wrap Up and Resources:**\ We ensure to link to:

* OpenCV’s Object Tracking API reference and pertinent blog posts for further reading[[3]](https://woteq.com/object-tracking-with-csrt-tracker-in-python-using-opencv/).
* The OTB dataset site if they want to try those videos (mention OTB has ground truth so one can quantitatively evaluate trackers).
* The MOT15/MOT17 datasets for multi-person tracking, if they are curious about the next level (multi-object).
* Possibly the **Norfair** library (MIT-licensed) as an easy Python tool for multi-object tracking by matching detections (if they want to explore an alternative, but this might be extra).

By completing Notebook 3, the student will:

* Know how to read and display video frames with OpenCV.
* Be able to initialize a tracker and update it through frames.
* Understand the basics of drawing on frames (rectangles, text).
* Experience issues like tracker loss, and know that different algorithms exist to mitigate those (giving a realistic sense of challenges in CV).
* They’ll have an interactive component (ROI selection) which makes the exercise more engaging.

A screenshot of a computer

AI-generated content may be incorrect.

**Notebook 4: Video-Based Movement Analysis**

**Goal:** Analyze the *type* or *characteristics* of movement in videos. Instead of just tracking position, we investigate *how* something is moving. This can include visualizing motion (with optical flow), quantifying motion (speed, direction consistency), or recognizing actions. We’ll keep it simple with an example of distinguishing different human actions via their motion patterns, leveraging classical techniques like optical flow and motion history. This gives the student a taste of action recognition without diving into heavy deep learning models.

**Data:** We can use the **KTH Actions dataset** (open access for research) which contains 6 actions performed by 25 people in controlled settings. For instance, we might include two short videos: one of someone walking and one of someone jogging (both from KTH). These are low-res (160×120) and grayscale, which is actually easier for optical flow and quick processing. If KTH data is not readily at hand, we could record two small clips ourselves (or use any free video of e.g. waving hand vs clapping). However, KTH is nice because it's a standard dataset: actions like walking, jogging, boxing are clearly differentiated.

Assume we have walking.avi and jogging.avi as example videos for this notebook.

**Step 1 - Load Video and Preprocess:**\ Similar to Notebook 3, open the video file. If it’s color, convert to grayscale because optical flow algorithms typically work on grayscale frames for efficiency.

cap = cv2.VideoCapture('walking.avi')

ret, prev\_frame = cap.read()

prev\_gray = cv2.cvtColor(prev\_frame, cv2.COLOR\_BGR2GRAY)

We also prepare a visualization canvas for optical flow:

h, w = prev\_gray.shape

# Create an image to draw flow vectors, initially all zeros (black)

flow\_visual = np.zeros((h, w, 3), dtype=np.uint8)

# We'll draw the flow vectors on this image as colored lines or arrows

*Markdown:* Explain that we took the first frame, converted to grayscale. We created flow\_visual as an RGB image filled with zeros, on which to draw flow. Alternatively, we might choose to visualize flow using the HSV color encoding trick (common for optical flow visualization)[[4]](https://www.geeksforgeeks.org/python/opencv-the-gunnar-farneback-optical-flow/). The code for that is a bit more complex, but we plan to show it later. We set the stage by introducing what optical flow is: *“Optical flow is the apparent motion of pixels between frames*[*[4]*](https://www.geeksforgeeks.org/python/opencv-the-gunnar-farneback-optical-flow/)*. If we compute it for each frame pair, we get a field of vectors indicating where each pixel moved.”*

**Step 2 - Compute Dense Optical Flow (Farneback):**\ Use OpenCV’s calcOpticalFlowFarneback to compute dense flow for consecutive frames[[4][4]](https://www.geeksforgeeks.org/python/opencv-the-gunnar-farneback-optical-flow/):

# Parameters for Farneback optical flow

fb\_params = dict(pyr\_scale=0.5, levels=3, winsize=15, iterations=3,

poly\_n=5, poly\_sigma=1.2, flags=0)

idx = 0

while True:

ret, frame = cap.read()

if not ret:

break

idx += 1

curr\_gray = cv2.cvtColor(frame, cv2.COLOR\_BGR2GRAY)

flow = cv2.calcOpticalFlowFarneback(prev\_gray, curr\_gray, None, \*\*fb\_params)

# Compute magnitude and angle of flow

mag, ang = cv2.cartToPolar(flow[...,0], flow[...,1])

# Use HSV image to represent flow (as in OpenCV docs)

hsv = np.zeros((h, w, 3), dtype=np.uint8)

hsv[...,1] = 255 # saturation to max

hsv[...,0] = ang \* 180 / np.pi / 2 # hue represents direction

hsv[...,2] = cv2.normalize(mag, None, 0, 255, cv2.NORM\_MINMAX)

flow\_rgb = cv2.cvtColor(hsv, cv2.COLOR\_HSV2BGR)

# Display or save some frames of flow\_rgb if needed

if idx % 10 == 0:

plt.imshow(flow\_rgb)

plt.title(f"Optical Flow at frame {idx}")

plt.axis('off')

plt.show()

prev\_gray = curr\_gray.copy()

cap.release()

*Markdown:* This is a bit dense, so we break it down:

* We set parameters for Farneback’s algorithm (explained: pyramid scale, window size, etc. – or mention these are default/balanced parameters).
* For each frame, we convert to grayscale curr\_gray.
* Compute flow which is a 2-channel float array of shape (h, w, 2) containing flow vectors (dx, dy) for each pixel[[4]](https://www.geeksforgeeks.org/python/opencv-the-gunnar-farneback-optical-flow/).
* We convert flow to polar coordinates: mag (speed of motion) and ang (direction of motion)[[4]](https://www.geeksforgeeks.org/python/opencv-the-gunnar-farneback-optical-flow/).
* We create an HSV image where hue encodes direction (0-180 in OpenCV Hue corresponds to 0-360 degrees, we scale angle accordingly) and value encodes magnitude (normalized to 0-255)[[4]](https://www.geeksforgeeks.org/python/opencv-the-gunnar-farneback-optical-flow/). Saturation is set to max for vivid colors.
* Convert HSV to BGR to get a human-viewable representation flow\_rgb.
* We show an example of the flow map every 10 frames (to avoid too much output). Each flow map is a psychedelic image where colors indicate direction of movement (legend: e.g., rightward flow might appear red, leftward blue, upward maybe cyan, depending on how hues map) and brightness indicates speed (bright = moving fast).

We’ll include in markdown a note: \*“Above, we visualize optical flow using the HSV color wheel approach[[4]](https://www.geeksforgeeks.org/python/opencv-the-gunnar-farneback-optical-flow/): hue = flow direction, brightness = flow magnitude. You can see that as the person walks, the background shows opposite motion vectors (since camera is static, background moves \*relative\* to person). The legs and arms of the walking person produce alternating color patterns because they move back-and-forth.”\* We might include a static legend image or simply describe: e.g., *“Green might indicate downward motion, blue leftward, etc.”*

* Also highlight that we update prev\_gray = curr\_gray at end to slide the window for the next pair comparison.

**Step 3 - Analyze Motion from Flow:**\ We can derive simple metrics from the flow:

* Compute average motion magnitude in each frame. For walking vs jogging, jogging frames should have higher average magnitude (faster motion).
* Compute the direction consistency: e.g., walking might have a more consistent left-to-right direction (if person is walking leftwards across frame) whereas boxing action might have more erratic, changing flow directions.

We can demonstrate a simple analysis:

cap = cv2.VideoCapture('walking.avi')

prev\_gray = None

motion\_magnitudes = []

while True:

ret, frame = cap.read()

if not ret:

break

curr\_gray = cv2.cvtColor(frame, cv2.COLOR\_BGR2GRAY)

if prev\_gray is not None:

flow = cv2.calcOpticalFlowFarneback(prev\_gray, curr\_gray, None, \*\*fb\_params)

mag, ang = cv2.cartToPolar(flow[...,0], flow[...,1])

motion\_magnitudes.append(mag.mean())

prev\_gray = curr\_gray

cap.release()

print("Average motion magnitude for walking video:", np.mean(motion\_magnitudes))

Then do the same for jogging.avi and compare:

# Repeat for jogging video...

print("Average motion magnitude for jogging video:", np.mean(motion\_magnitudes\_jogging))

*Markdown:* We expect the jogging video to yield a higher average flow magnitude than walking (since limbs and body move faster). We’ll phrase an observation: *“We found the mean optical flow magnitude for walking ~ X, and for jogging ~ Y, confirming jogging has faster motion.”* This simplistic metric can serve as an “action recognition” criterion in a trivial sense (distinguish slow vs fast actions). We clarify that real action recognition would involve more sophisticated pattern recognition (could mention deep learning models like 3D ConvNets or Two-Stream Networks which use optical flow as input).

**Step 4 - Motion History Image (MHI) (optional):**\ Another simple movement analysis tool is the **Motion History Image**, which accumulates recent motion in an image. We can introduce this if time permits:

* Compute frame differencing: motion\_mask = cv2.absdiff(curr\_gray, prev\_gray) threshold it to get where movement occurred.
* Accumulate in an MHI: mhi = np.where(motion\_mask>thresh, timestamp, decrease\_old\_values). OpenCV has a function cv2.motempl.updateMotionHistory to do this. If included:
  + Show the silhouette of motion over last second as a gradient image (recent motion bright, older motion dark).
  + This allows seeing the *path* of motion. For example, a hand waving might produce a fuzzy arc shape in the MHI, whereas running yields a stretched silhouette.

This might be too advanced, but at least mention it: *“We can also create a motion history image (MHI) – a static image summarizing movement over time. OpenCV’s motion module (motempl) provides functions for this. This can be used for action templates (as in the famous paper by Bobick and Davis). See OpenCV’s sample motempl.c for an example.”* (We won’t implement fully due to time, but provide the concept and reference).

**Step 5 - (Optional) Simple Action Classifier:**\ If we want a final challenge: use optical flow results to classify an unknown video as “walking” or “jogging”:

* We could extract a feature like average flow magnitude or frequency content of motion (e.g., count oscillations).
* Example: the leg movement in walking vs running has different frequency. Maybe count zero-crossings in optical flow in a region (though this is complex to do robustly).

Given this is advanced, we might just describe it: *“One could differentiate actions by analyzing the periodicity of motion. Walking typically ~ 1-2 steps per second, running ~ 3-4 steps per second. If we tracked a point on the leg using optical flow or just measured frame-to-frame changes, we could estimate this frequency. However, reliably doing this can be tricky; this is where machine learning would come in – to learn patterns of motion.”*

**Wrap-up Discussion and Resources:**\ Finally, provide context and next steps:

* *“Optical flow is useful not just for visualization, but as input to action recognition algorithms. For example, the Two-Stream CNN model (Simonyan & Zisserman) takes both RGB frames and optical flow as input to classify actions. Modern approaches like 3D ConvNets (e.g., I3D, SlowFast networks) operate on raw video.”* – We won’t implement these, but we give references:
  + Link to a GitHub with pretrained action models or a TensorFlow Hub model for video classification (e.g., mentioning that one could try the TF Hub “MoveNet” for pose or an I3D model for classification on something like UCF101).
  + If KTH was used, mention that simple methods can actually classify KTH with decent accuracy (as per research ~90% with HOG+HOF features). We can link a resource where someone implemented a classic solution (maybe the GitHub we found in the plan: vkhoi’s KTH Action Recognition which used optical flow + bag of words).
* Also encourage the student to try their own videos: *“Try recording yourself doing two different actions and see if you can spot differences in the optical flow or other measures.”*

**Extra notes:** We ensure any data used is open: KTH is free for research (we cite its info: 6 actions, 25 people, 2391 clips). If we include KTH videos in the training materials, that's allowed for non-commercial/educational use with credit (perhaps include a note "Courtesy of KTH Dataset"). Alternatively, use something like UCF101 videos (but those are bigger and more complex). KTH is simpler.

By finishing Notebook 4, the student will:

* Know how to compute and interpret optical flow.
* Have an appreciation for how motion can be visualized and quantified.
* Understand that distinguishing actions involves looking at motion patterns (and that one can start with simple stats or go into advanced ML).
* Be aware of more advanced concepts like motion history images and deep action recognition models, with pointers to learn more.

**Final Tips:** Across all notebooks, we will use plenty of **markdown commentary** and **links**:

* Each step is preceded by a markdown cell explaining the purpose.
* Key functions (cv2.threshold, cv2.findContours, cv2.calcOpticalFlowFarneback, cv2.TrackerCSRT\_create, etc.) will be explained with references to OpenCV docs or tutorials.
* We’ll integrate small challenges or questions in markdown, like “Try adjusting the threshold value and see what happens” or “What if we track a different object, or use a different tracker? (Change one line and observe)”.
* We aim for an interactive tone, encouraging the student to tweak parameters and witness the effect, reinforcing learning by doing.

Each notebook stands alone but is part of the larger progression. By completing these, the student will have a solid practical foundation in computer vision:

1. Classification taught them about modeling and basic image data handling.
2. Segmentation taught image processing and pixel-level reasoning.
3. Tracking taught video handling and dynamic analysis of position.
4. Movement analysis taught extraction of temporal patterns and advanced use of OpenCV functions.

With these skills, the student will be well-prepared to tackle the **CeleST worm movement project**, which combines elements of segmentation (finding worm outlines), tracking (following worms), and movement analysis (measuring motion features). They’ll also be comfortable reading documentation and using open-source tools, as we have consistently linked to resources and demonstrated how to use them. Each notebook can be run independently and contains everything needed (data loading code or small attached files), ensuring an **accessible, hands-on learning experience**.

**References**

[1] [Simple Digit Recognition OCR in OpenCV-Python - Stack Overflow](https://stackoverflow.com/questions/9413216/simple-digit-recognition-ocr-in-opencv-python)

[2] [Interactive Foreground Extraction using GrabCut Algorithm](https://docs.opencv.org/3.4/d8/d83/tutorial_py_grabcut.html)

[3] [Object Tracking with CSRT Tracker in Python using OpenCV](https://woteq.com/object-tracking-with-csrt-tracker-in-python-using-opencv/)

[4] [OpenCV - The Gunnar-Farneback optical flow - GeeksforGeeks](https://www.geeksforgeeks.org/python/opencv-the-gunnar-farneback-optical-flow/)