st125981_CV_A2_Task_3

November 1, 2025

0.1 CV Assignment 2 - Task 3

Name: Muhammad Fahad Waqar Student No: st125981

[2]:
```
import cv2
import time
import os
import requests
import numpy as np
import pandas as pd
```

[6]:
```
OPENCV_TRACKERS = {
    "csrt": cv2.legacy.TrackerCSRT_create,
    "kcf": cv2.legacy.TrackerKCF_create,
    "mosse": cv2.legacy.TrackerMOSSE_create
}

TRACKERS_TO_TEST = ["kcf", "csrt"]

# VIDEO_FILENAME = "videos/2103099-uhd_3840_2160_30fps.mp4"
VIDEO_FILENAME = "videos/853960-hd_1920_1080_25fps.mp4"
```

[11]:
```
def run_tracking_session(tracker_name, video_path):
    # Get the tracker constructor from our dictionary
    tracker_constructor = OPENCV_TRACKERS.get(tracker_name)
    if not tracker_constructor:
        print(f"Error: Tracker '{tracker_name}' not found in OpenCV version.")
        print("Please ensure you have 'opencv-contrib-python' installed.")
        return [], 0

    try:
        tracker = tracker_constructor()
    except cv2.error as e:
        print(f"Failed to create tracker '{tracker_name}'.")
        print("Error: {e}")
        print("Please ensure you have 'opencv-contrib-python' installed␣
 ↪correctly.")
        return [], 0
    # Open the video
    cap = cv2.VideoCapture(video_path)
    if not cap.isOpened():
        print(f"Error: Could not open video file {video_path}")
        print("Please check the path in VIDEO_FILENAME.")
        return [], 0

    # Read the first frame
    ok, first_frame = cap.read()
    if not ok:
        print("Error: Could not read first frame.")
        cap.release() # Add release
        return 0, 0, 0

    # Select ROI (Region of Interest)
    print("\n" + "-"*50)
    print(f"Running Tracker: {tracker_name.upper()}")
    print("A window will pop up. ")
    print("1. Use your mouse to draw a box around the object you want to track.")
    print("2. Press ENTER or SPACE to confirm.")
    print("3. Press 'c' to cancel selection.")

    # FIX for High-Resolution Screens
    max_display_width = 1280
    orig_h, orig_w = first_frame.shape[:2]

    # Only resize if the original width is larger than our max display width
    if orig_w > max_display_width:
        scale_factor = max_display_width / orig_w
        display_w = int(orig_w * scale_factor)
        display_h = int(orig_h * scale_factor)
        display_frame = cv2.resize(first_frame, (display_w, display_h))
    else:
        # If it's already small enough, just use the original
        display_frame = first_frame
        scale_factor = 1.0

    # selectROI returns (x, y, w, h)
    init_bbox_scaled = cv2.selectROI("Select Object to Track", display_frame, fromCenter=False, showCrosshair=True)
    cv2.destroyWindow("Select Object to Track")

    if init_bbox_scaled == (0, 0, 0, 0):
        print("No bounding box selected. Aborting session.")
        cap.release()
```

```python
        return 0, 0, 0

    # Scale the bounding box back to the original frame's dimensions
    inv_scale_factor = 1.0 / scale_factor
    init_bbox = tuple([int(v * inv_scale_factor) for v in init_bbox_scaled])

    # Initialize Tracker
    tracker.init(first_frame, init_bbox)

    # Setup Output Video
    frame_width = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
    frame_height = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
    video_fps = int(cap.get(cv2.CAP_PROP_FPS))
    output_filename = f"output_{tracker_name}.mp4"

    # Define the codec and create VideoWriter object
    # Using 'mp4v' for .mp4 files
    fourcc = cv2.VideoWriter_fourcc(*'mp4v')
    out = cv2.VideoWriter(output_filename, fourcc, video_fps, (frame_width, frame_height))

    print(f"Tracking started. Output will be saved to '{output_filename}'")

    fps_list = []
    failure_count = 0

    while True:
        ok, frame = cap.read()
        if not ok:
            # End of video
            break

        # Update Tracker
        start_time = time.time()
        success, bbox = tracker.update(frame)
        end_time = time.time()

        # Calculate FPS
        fps = 1.0 / (end_time - start_time)
        fps_list.append(fps)

        # Draw Results on Frame
        if success:
            # Tracking success
            (x, y, w, h) = [int(v) for v in bbox]
            cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 255, 0), 2, 1) # Green
            status_text = "STATUS: Tracking"
            status_color = (0, 255, 0)
        else:
            # Tracking failure
            failure_count += 1
            status_text = "STATUS: Tracking Failure"
            status_color = (0, 0, 255) # Red

        # Add info text to the frame
        info_text = f"Tracker: {tracker_name.upper()} | FPS: {int(fps)}"
        # cv2.putText(frame, info_text, (10, 20), cv2.FONT_HERSHEY_SIMPLEX, 0.
↪6, (0, 0, 0), 2) # Black outline
        # cv2.putText(frame, info_text, (11, 21), cv2.FONT_HERSHEY_SIMPLEX, 0.
↪6, (255, 255, 255), 1) # White text

        # cv2.putText(frame, status_text, (10, 45), cv2.FONT_HERSHEY_SIMPLEX, 0.
↪6, (0, 0, 0), 2)
        # cv2.putText(frame, status_text, (11, 46), cv2.FONT_HERSHEY_SIMPLEX, 0.
↪6, status_color, 1)

        font_scale = 1.2
        font_thickness_outline = 3
        font_thickness_text = 2

        info_pos = (20, 40)  # (x, y) for top line
        status_pos = (20, 80) # (x, y) for bottom line

        cv2.putText(frame, info_text, info_pos, cv2.FONT_HERSHEY_SIMPLEX,␣
↪font_scale, (0, 0, 0), font_thickness_outline) # Black outline
        cv2.putText(frame, info_text, (info_pos[0]+1, info_pos[1]+1), cv2.
↪FONT_HERSHEY_SIMPLEX, font_scale, (255, 255, 255), font_thickness_text) #␣
↪White text

        cv2.putText(frame, status_text, status_pos, cv2.FONT_HERSHEY_SIMPLEX,␣
↪font_scale, (0, 0, 0), font_thickness_outline)
        cv2.putText(frame, status_text, (status_pos[0]+1, status_pos[1]+1), cv2.
↪FONT_HERSHEY_SIMPLEX, font_scale, status_color, font_thickness_text)

        # Write Frame to Output Video
        out.write(frame)

        # Display Resized Frame
        # Resize the frame for display to prevent cropping on large videos
        if orig_w > max_display_width:
            # We use the display_w and display_h calculated during ROI selection
            display_frame_live = cv2.resize(frame, (display_w, display_h))
        else:
            display_frame_live = frame

        # Display the resized frame
        cv2.imshow(f"Tracking with {tracker_name.upper()}", display_frame_live)

        # Exit if 'q' is pressed
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

    # Cleanup
    print(f"Tracking session for {tracker_name.upper()} finished.")
    cap.release()
    out.release()
    cv2.destroyAllWindows()

    # Calculate average FPS and return
    avg_fps = np.mean(fps_list) if fps_list else 0
    return avg_fps, failure_count, len(fps_list)
```

[13]:

```
# Check if video exists
if not os.path.exists(VIDEO_FILENAME) or os.path.getsize(VIDEO_FILENAME) == 0:
    print(f"Error: Sample video '{VIDEO_FILENAME}' is missing or empty.")
    print("Please make sure the VIDEO_FILENAME variable in Cell [2] is set to␣
 ↪the correct path.")
else:
    # Run tracking sessions for each tracker
    results = {}
    total_frames = 0

    for tracker_name in TRACKERS_TO_TEST:
        avg_fps, failures, num_frames = run_tracking_session(tracker_name,␣
 ↪VIDEO_FILENAME)

        if num_frames > 0:
            total_frames = num_frames
            results[tracker_name] = {
                "avg_fps": avg_fps,
                "failures": failures,
                "success_rate": 100 * (num_frames - failures) / num_frames
            }
        else:
            print(f"Session for {tracker_name} was cancelled or failed to start.
 ↪")

    # Print results
    if results:
        # Print a formatted table
        print(f"{'Metric':<15} | {'KCF':<20} | {'CSRT':<20} |")
        print(f"{'-'*15} | {'-'*20} | {'-'*20} |")

        kcf_fps = results.get("kcf", {}).get("avg_fps", 0)
        csrt_fps = results.get("csrt", {}).get("avg_fps", 0)
        print(f"{'Avg. FPS':<15} | {kcf_fps:<20.2f} | {csrt_fps:<20.2f} |")

        kcf_fail = results.get("kcf", {}).get("failures", 0)
        csrt_fail = results.get("csrt", {}).get("failures", 0)
        print(f"{'Total Failures':<15} | {kcf_fail:<20} | {csrt_fail:<20} |")

        kcf_rate = results.get("kcf", {}).get("success_rate", 0)
        csrt_rate = results.get("csrt", {}).get("success_rate", 0)
        print(f"{'Success Rate':<15} | {kcf_rate:<19.2f}% | {csrt_rate:<19.2f}% |")

    else:
        print("No tracking sessions were completed.")
```

```
--------------------------------------------------
Running Tracker: KCF
A window will pop up.
1. Use your mouse to draw a box around the object you want to track.
2. Press ENTER or SPACE to confirm.
3. Press 'c' to cancel selection.
Tracking started. Output will be saved to 'output_kcf.mp4'
Tracking session for KCF finished.

--------------------------------------------------
Running Tracker: CSRT
A window will pop up.
1. Use your mouse to draw a box around the object you want to track.
2. Press ENTER or SPACE to confirm.
3. Press 'c' to cancel selection.
Tracking started. Output will be saved to 'output_csrt.mp4'
Tracking session for CSRT finished.
Metric          | KCF                  | CSRT                 |
--------------- | -------------------- | -------------------- |
Avg. FPS        | 207.19               | 31.62                |
Total Failures  | 125                  | 0                    |
Success Rate    | 52.83              % | 100.00             % |
```

0.2 Discussion

0.3 Tracker Architecture Comparison

0.3.1 KCF

- Operates in the Fourier domain, enabling fast correlation computations using FFT.

- Uses kernel tricks to build a non-linear classifier that separates the object from the background.

- Learns a correlation filter from the target's appearance and applies it frame-by-frame to locate the object.

- Prioritizes speed and efficiency but struggles with large appearance changes or occlusions.

0.3.2 CSRT

- A more advanced discriminative correlation filter tracker.

- Uses spatial reliability maps and multiple feature channels (color, gradient, etc.).

- Assigns reliability weights to different regions and channels, improving robustness to occlusion and appearance changes.

- Focuses on accuracy and stability at the cost of speed.

0.3.3 Performance Analysis

0.3.4 1. Tracking Speed (FPS)

Tracker	FPS	Relative Speed
KCF	23.62	—
CSRT	7.09	**3.3× slower**

- **KCF** achieves higher speed through **FFT-based computation**, ideal for **real-time tracking**.

- **CSRT**'s multi-channel analysis and reliability computation increase accuracy but **reduce frame rate**.

0.3.5 2. Tracking Robustness

Metric	KCF	CSRT
Success Rate	100%	100%
Stability	Slight jitter during movement	Smoother, more precise tracking

- Both trackers succeeded in this controlled video, indicating **favorable conditions** (no major occlusions or drastic changes).

- **KCF:** Minor jitter and drift during rapid motion or lighting changes.

- **CSRT:** Maintained tighter bounding box alignment and smoother tracking throughout.

0.3.6 3. Failure Cases and Limitations

KCF Weaknesses

- **Scale Variation:** Fixed-size filter → poor adaptability to large scale changes.

- **Appearance Changes:** Sensitive to rotation and pose variations.

- **Occlusion:** Limited robustness to partial or full occlusions.

- **Background Clutter:** May drift if background patterns resemble the target.

CSRT Weaknesses

- **Computational Cost:** Slower due to feature weighting and spatial reliability mapping.

- **Complete Occlusions:** More robust than KCF but still prone to long-term failures.

- **Initialization Sensitivity:** Requires precise initial bounding box for accurate tracking.

0.3.7 Key Trade-offs (For the tested video)

Aspect	KCF	CSRT
Speed	Faster (23.6 FPS)	Slower (7.1 FPS)
Accuracy	Moderate	High
Occlusion Handling	Weak	Stronger
Adaptability	Limited	Better (spatial reliability)
Resource Efficiency	Lightweight	Computationally heavy

0.3.8 Comparison with Modern Deep Learning Trackers

Aspect	Classical Trackers (KCF/CSRT)	Deep Trackers (e.g., SiamFC, DiMP, ATOM)
Appearance Adaptation	Limited	Excellent
Occlusion Handling	Moderate	Strong
Scale Handling	Weak (KCF) / Moderate (CSRT)	Robust

Aspect	Classical Trackers (KCF/CSRT)	Deep Trackers (e.g., SiamFC, DiMP, ATOM)
Resource Requirement	Low	High (GPU, training data)
Ease of Use	Plug-and-play	Requires training & dependencies

- Deep trackers provide **superior accuracy and robustness** but require GPUs and pretrained models.

- Classical trackers remain **relevant for lightweight, quick deployment**—especially when training data or high-end hardware is unavailable.