Assignment 6 - Condensation Tracker

Federica Bruni (fbruni@student.ethz.ch) 6 December 2023

1 Overview

This lab aims to implement a CONDENSATION tracker focusing on tracking a single object represented by bounding boxes with states including location and velocities. Two prediction models are considered: no motion and constant velocity. The tracker relies on color histograms to assess similarity, utilizing the x^2 distance and assigning weights to particles. The user provides the initial bounding box, and the algorithm prioritizes particles with color histograms similar to the target model. The lab report will elaborate on the implementation and analysis of this tracking approach, emphasizing the experimental process of parameter tuning determining their effect on tracking performance.

2 CONDENSATION Tracker Based On Color Histograms

2.1 Color histrogram

To address the task of calculating the normalized histogram of RGB colors within a specified bounding box, I implemented the *color_histogram* function.

This function extracts the region of interest from the frame using the provided coordinates. Subsequently, it separates the ROI into its individual RGB channels. For each channel an histogram is computed, ensuring that the values fall within the specified bin range. These individual histograms are concatenated into a single histogram, representing the color distribution within the bounding box. Finally, the histogram is normalized by dividing each bin count by the total count, resulting in a probability distribution that characterizes the color content within the specified region.

```
def color_histogram(xmin, ymin, xmax, ymax, frame, hist_bin):
    # the region of interest from the frame
    roi = frame[ymin:ymax, xmin:xmax, :]

    # split into RGB channels
    r = roi[:, :, 0]
    g = roi[:, :, 1]
    b = roi[:, :, 2]

    # histogram values for each channel
    r_hist, r_edges = np.histogram(r.flatten(), bins=hist_bin, range=[0, 256])
    g_hist, g_edges = np.histogram(g.flatten(), bins=hist_bin, range=[0, 256])
    b_hist, b_edges = np.histogram(b.flatten(), bins=hist_bin, range=[0, 256])

    # concatenate all histograms
    hist = np.concatenate((r_hist, g_hist, b_hist))

    # normalize the histogram
    hist = hist / np.sum(hist)

    return hist
```

2.2 Propagation

To fulfill the task of particle propagation based on the system prediction model and system model noise, I implemented the *propagate* function. This function extracts the number of particles and the state dimensionality from the particle array. The function then determines the prediction model to apply. If the model is set to 0, indicating no motion, an identity matrix A is used, and particles are updated by matrix multiplication with random noise generated from a normal distribution. If the model is set to 1, representing constant velocity, a predefined matrix A is used, and particles are updated with additional

noise for both position and velocity. After the particle updates, the function ensures that the particles remain within the frame boundaries.

```
def propagate(particles, frame_height, frame_width, params):
    n_particles, dim = particles.shape

    if params['model'] == 0:  # no motion
        A = np.eye(dim)
        # update particle positions using the no motion model and add random noise
        particles = A @ particles.T + np.random.randn(dim, n_particles) * params['
    sigma_position']
        particles = particles.T
    else:  # constant velocity
        # system matrix for constant velocity motion model
        A = np.array([[1, 0, 1, 0],
                      [0, 1, 0, 1],
                      [0, 0, 1, 0],
                      [0, 0, 0, 1]])
        # random noise for position and velocity
        noise_position = np.random.randn(2, n_particles) * params['sigma_position']
        noise_velocity = np.random.randn(2, n_particles) * params['sigma_velocity']
        # update particle positions using the constant velocity motion model and noise
        particles = A @ particles.T + np.vstack((noise_position, noise_velocity))
        particles = particles.T

    # check that the particles lie inside the frame
    particles[:, 0] = np.clip(particles[:, 0], 0, frame_width)
    particles[:, 1] = np.clip(particles[:, 1], 0, frame_height)

    return particles
```

2.3 Observation

The *observe* function is designed to perform observations on a set of particles. The function computes color histograms for each particle within its respective bounding box. It utilizes the previously defined *color_histogram* function for this purpose. Subsequently, the Chi-squared distance between the histogram of each particle and the target histogram is calculated using the provided *chi2_cost* function. The weights of the particles are then updated based on the Chi-squared distances and the observation noise. A Gaussian distribution is employed to model the likelihood of each particle's color histogram matching the target histogram. The weights are computed using the Gaussian probability density function, and they are normalized to ensure a valid probability distribution.

```
def observe(particles, frame, bbox_height, bbox_width, hist_bin, hist_target,
    sigma_observe):
    height_frame, width_frame, _ = frame.shape
    n_particles = len(particles)
    particles_w = np.zeros(n_particles)

    # color histograms for all particles
    hist_particles = np.zeros((n_particles, hist_bin * 3))
    for i in range(n_particles):
        # bounding box around the particle center
        xmin = max(0, int(np.floor(particles[i, 0] - 0.5 * bbox_width)))
        ymin = max(0, int(np.floor(particles[i, 1] - 0.5 * bbox_height)))
        xmax = min(width_frame, int(np.floor(particles[i, 0] + 0.5 * bbox_width)))
        ymax = min(height_frame, int(np.floor(particles[i, 1] + 0.5 * bbox_height)))

        # color histogram for the particle within the bounding box
        hist_particles[i, :] = color_histogram(xmin, ymin, xmax, ymax, frame, hist_bin)

        # chi2 distance between the particle histogram and the target histogram
        chi2_distance = chi2_cost(hist_target, hist_particles[i, :])

        # update the weight of the particle using a Gaussian distribution
        particles_w[i] = (1 / (np.sqrt(2 * np.pi) * sigma_observe)) * \
                         np.exp(-0.5 * chi2_distance**2 / sigma_observe**2)

    # normalize particle weights
    particles_w = particles_w / np.sum(particles_w)

    return particles_w
```

2.4 Estimation

The *estimate* function is implemented to compute the mean state based on a set of particles and their associated weights. The function calculates the mean state by taking the dot product of the transposed particles array and the weight vector.

```
def estimate(particles, particles_w):
    mean_state = np.dot(particles.T, particles_w)
    return mean_state.T
```

2.5 Resampling

The *resample* function is designed to implement the step of resampling particles based on their weights. The function utilizes a systematic resampling approach, generating a random threshold and systematically selecting particles based on cumulative weights. The loop iterates through each particle, and for each iteration, a threshold U is calculated to determine the next selected particle. The cumulative weight c is updated until it surpasses the threshold, at which point the corresponding particle is added to the new set of particles. The process ensures that particles with higher weights have a higher likelihood of being selected. The resulting set of resampled particles and their corresponding weights are then normalized to maintain a valid probability distribution.

```
def resample(particles, particles_w):
    n_particles = len(particles)

    # random number in the range [0, 1) for resampling
    r = 1 / n_particles * np.random.rand()

    c = particles_w[0]
    i = 0
    new_particles = []
    new_particles_w = []

    # resampling loop
    for m in range(1, n_particles + 1):
        # threshold for selecting the next particle
        U = r + 1 / n_particles * (m - 1)

        # update the index until the cumulative weight exceeds the threshold
        while U > c:
            i += 1
            c += particles_w[i]
        new_particles.append(particles[i, :])
        new_particles_w.append(particles_w[i])

    # normalize the new particle weights
    new_particles_w = np.array(new_particles_w) / np.sum(new_particles_w)

    return np.array(new_particles), new_particles_w
```

3 Experiments

Next we are presented with three distinct videos—video1.wmv, video2.wmv, and video3.wmv—each showcasing a moving object under different environmental conditions.

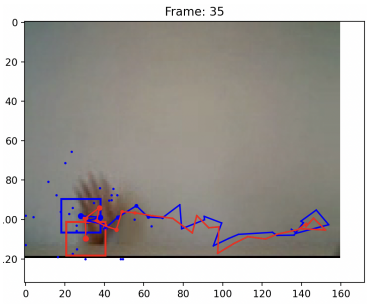
1. The first video, video1.wmv, features a moving hand against a uniform background.
2. The second video, video2.wmv, presents a moving hand with some clutter and occlusions.
3. video3.wmv displays a bouncing ball.

The objective is to track these objects across the videos. The tracking algorithm, guided by a range of parameters will undergo systematic experimentation to evaluate its performance across diverse scenarios. The goal is to figure out how changing these settings affects how accurately and reliably the tracker follows objects. This step-by-step exploration of different setups will give us a clearer picture of the best choices for each video scenario, making the tracker more effective in different situations.

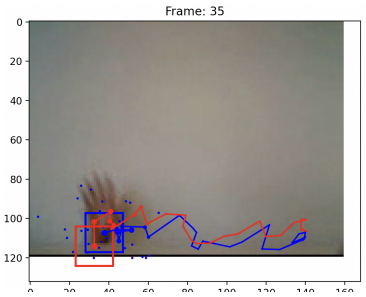
3.1 Moving hand with a uniform background

3.1.1 Default parameters

Prior to starting experiments on video 1, I present the results obtained with the default parameters for both models: model0 (no-motion) and model1 (constant velocity). Below you can find both the table reporting the paramterets used and the resulting tracking paths.



(a) model 0



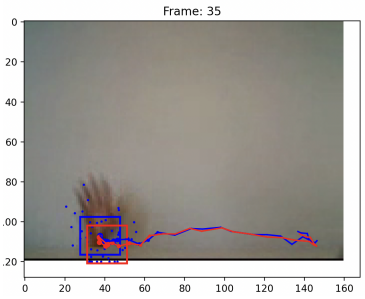
(b) model 1

Default parameters	
Parameter	Value
draw_plots	1
hist_bin	16
alpha	0
sigma_observe	0.1
model	0/1
num_particles	30
sigma_position	15
sigma_velocity	1
initial_velocity	(1,10)

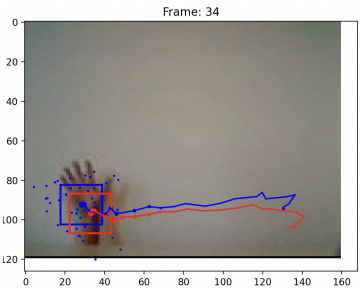
The obtained results demonstrate that the implemented tracker achieves good performance and successfully tracks the hand. However, the main issue is its tendency to lose focus on the hand and concentrate on the wrist. This occurs due to lighting changes in the video, where initially shadowed fingers become illuminated and change color, while the wrist consistently remains in shadow, leading the tracker to prioritize it. Additionally, as evident from the trace of both prior and posterior mean states, there are numerous oscillations in the tracking process.

3.1.2 Final paramterers choice

Prior to starting experiments on video 1, I present the results obtained with the default parameters for both models: model0 (no-motion) and model1 (constant velocity). Below you can find both the table reporting the paramterets used and the resulting tracking paths.



(a) model 0



(b) model 1

Final Parameters video1	
Parameter	Value
draw_plots	1
hist_bin	48
alpha	0.7
sigma_observe	0.2
model	0/1
num_particles	50
sigma_position	8
sigma_velocity	1
initial_velocity	(-5,-10)

Modifying the tracking algorithm parameters from the original configuration to the updated settings has significantly contributed to enhancing the stability and robustness of the tracker, particularly in the face of lighting changes.

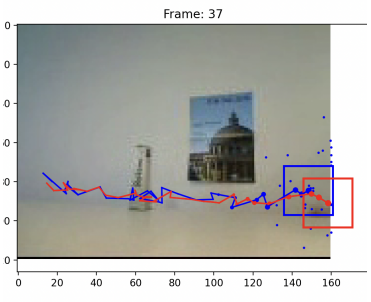
- The increased number of bins in the color histogram (hist_bin: 16 to 48) enables a more detailed characterization of color distributions, allowing the tracker to better adapt to variations in lighting conditions.
- Adjusting the weight threshold parameter (alpha: 0 to 0.7) influences the importance assigned to new observations, facilitating a more adaptive response to changes.
- The augmentation of the number of particles (num_particles: 30 to 50) enhances the representation of the posterior distribution, providing a more comprehensive exploration of the state space and making the tracker less susceptible to outliers.
- Fine-tuning the standard deviation of observation noise (sigma_observe: 0.1 to 0.2) contributes to a more accurate modeling of uncertainty, crucial for handling varying lighting scenarios.

- Further, modifying the initial velocity of the tracked object (initial_velocity: (1, 10) to (-5, -10)) allows the algorithm to better initialize the motion model.

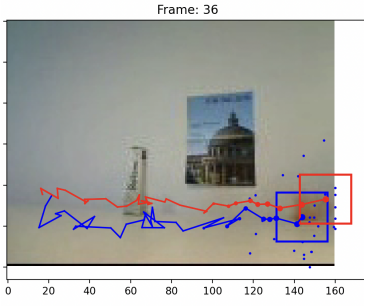
3.2 Moving hand with some clutter and occlusions

3.2.1 Default parameters

Continuing with video 2, I present the results obtained using the default parameters for both model 0 and model 1.



(a) model 0

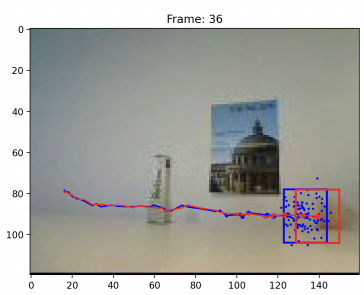


(b) model 1

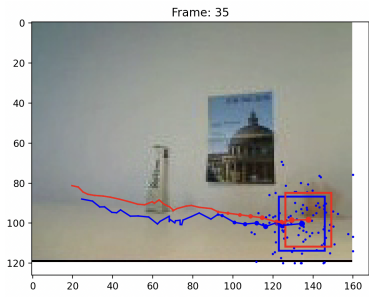
Default parameters	
Parameter	Value
draw_plots	1
hist_bin	16
alpha	0
sigma_observe	0.1
model	0/1
num_particles	30
sigma_position	15
sigma_velocity	1
initial_velocity	(1,10)

I was impressed by the tracker's accuracy despite the occlusions. Unlike the first video, there are no changes in lighting here, making the tracker more accurate. However, it can still be further stabilized with a fine-tuning process on the parameters.

3.2.2 Final paramterers choice



(a) model 0



(b) model 1

Final parameters video2	
Parameter	Value
draw_plots	1
hist_bin	32
alpha	0.4
sigma_observe	0.1
model	0/1
num_particles	100
sigma_position	10
sigma_velocity	1
initial_velocity	(10,10)

Adjusting the parameters of the tracking algorithm from the initial configuration to the updated values has notably improved the tracker's stability, especially when faced with occlusion challenges.

- Increasing the number of histogram bins (hist_bin: 16 to 32) allows for a more detailed characterization of color distributions, aiding the tracker in maintaining accuracy during occlusion periods.
- A slight adjustment in the weight threshold (alpha: 0 to 0.4) influences the algorithm's sensitivity to new observations, offering a more balanced response to partial occlusions.
- Substantially increasing the number of particles (num_particles: 30 to 100) enhances the robustness of the tracker, facilitating better representation of the posterior distribution and enabling more accurate tracking even when parts of the object are temporarily hidden.
- Fine-tuning the standard deviation of particle position noise (sigma_position: 15 to 10) further contributes to the tracker's resilience by allowing particles to explore the state space more effectively during occlusion events.
- Additionally, modifying the initial velocity of the tracked object (initial_velocity: (1, 10) to (10, 10)) aids in predicting the object's trajectory more reliably, improving tracking continuity in the presence of occlusions.

Answers

- *What is the effect of using a constant velocity motion model?*

  Employing a constant velocity motion model in the context of a video featuring a moving hand with clutter and occlusion can have both advantages and limitations. A constant velocity model assumes a consistent and linear motion trajectory for the tracked object over time. In the case of a moving hand, this can contribute to smoother and more predictable predictions of the hand's future positions, aiding in the continuity of tracking. However, challenges may arise in scenarios like this one involving clutter and occlusion. The constant velocity model might struggle to adapt when the hand is temporarily hidden or obstructed by other objects, leading to potential inaccuracies in predicting the hand's location during occlusion.

- *What is the effect of assuming decreased/increased system noise?*

  In the context of tracking a hand in a video with clutter and occlusion, the adjustment of system noise parameters, specifically sigma_position and sigma_velocity, holds a key role in shaping the performance of the tracking algorithm.

  Increasing sigma_position enhances the filter's adaptability to position changes. However if sigma_position is too large, the particle cloud can spread too much, making the tracking less precise. Conversely, reducing sigma_position sharpens tracking precision within a smaller region. Yet, an excessively low value may limit the filter's ability to catch up with the object.

  Moreover, the adjustment of sigma_velocity contributes significantly to tracking accuracy. A decrease in sigma_velocity implies greater confidence in the motion model, leading to more precise predictions of the hand's movement. This proves advantageous in scenarios characterized by relatively steady and measured hand motion. Lower sigma_velocity values prevent particles from deviating significantly from the predicted path, focusing tracking on a narrower area. However, excessively low sigma_velocity might lead to motion complexity underestimation, resulting in tracking errors. Conversely, an increased sigma_velocity grants greater adaptability to unexpected motion, allowing particles to explore a wider state space.

  The delicate balance between sigma_velocity and system noise adjustments is crucial for optimizing tracking accuracy across diverse scenarios, particularly those involving clutter and occlusion.

- *What is the effect of assuming decreased/increased measurement noise?*

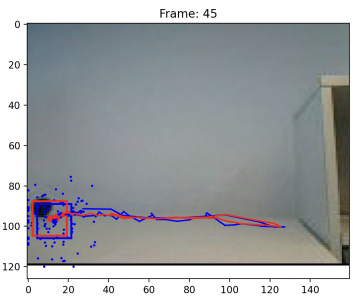
  When it comes to measurement noise, a reduction in sigma_observe instills greater confidence in the observed data, fostering improved reliance on reliable information for tracking accuracy. However, the caveat lies in the possibility that excessive trust in these observations, especially in the presence of occlusions or background objects, could lead to erroneous tracking.

  Conversely, incresing sigma_observe we obtain a more tolerant filter, proving beneficial in navigating the complexities introduced by background objects or occlusions.

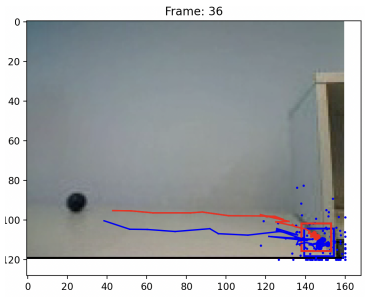
3.3 Ball bouncing

3.3.1 Final parameters from video2

As requested, here are the results of applying the final parameters from video 2 to video 3.



(a) model 0

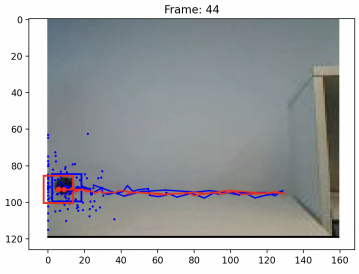


(b) model 1

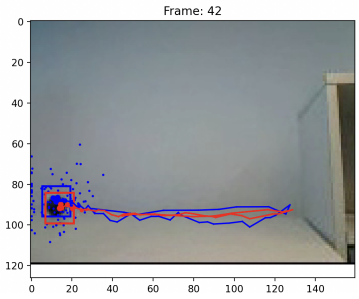
Final parameters video2	
Parameter	Value
draw_plots	1
hist_bin	32
alpha	0.4
sigma_observe	0.1
model	0/1
num_particles	100
sigma_position	10
sigma_velocity	1
initial_velocity	(10,10)

The obtained results demonstrate that with the model 0, the tracker is still able to track the ball. However, with model 1, the ball is lost after the bounce, and the tracker focuses on a dark corner with colors similar to the ball.

3.3.2 Final paramterers choice



(a) model 0



(b) model 1

Final parameters video3	
Parameter	Value
draw_plots	1
hist_bin	32
alpha	0.2
sigma_observe	0.1
model	0/1
num_particles	100
sigma_position	10
sigma_velocity	2
initial_velocity	(1,0)

The adjusted parameters in the second set played a crucial role in enhancing tracking performance in the scenario of a ball bouncing against a wall.

- The decrease in the weighting factor (alpha: 0.4 to 0.2) enables the algorithm to place greater emphasis on the observed data, fostering dynamic adjustments to the unexpected trajectory shifts during ball-wall interactions.
- Increasing sigma_velocity (sigma_velocity: 1 to 2) introduces more adaptability to the unexpected motion of the bouncing ball. A higher sigma_velocity allows the particles to explore a wider range of possible states, making the tracker more robust in capturing abrupt changes in the ball's velocity during bouncing.
- Adjusting the initial_velocity parameter from (10, 10) to (1, 0) contributes to more accurate predictions of the ball's trajectory. This adjustment aids in providing a more reliable starting point for the tracking process, reducing prediction errors and enhancing the tracker's stability.

Answers

- *What is the effect of using a constant velocity motion model?*

  The constant velocity model assumes that the object's motion remains uniform over time. However, the primary challenge arises during the impact with the wall, where the ball undergoes a change in velocity. The constant velocity model struggles to accurately predict the ball's trajectory during this dynamic event, leading to tracking errors.

  In summary, the constant velocity motion model works well for tracking the ball's overall horizontal motion but fails when the ball bounces off the wall due to the abrupt change in velocity and direction.

- *What is the effect of assuming decreased/increased system noise?*

  A reduced sigma_position, theoretically, enhances tracking precision by focusing particle spread in a smaller region. However, an excessively low value might hinder the model from keeping up with the ball's dynamic motion, leading to the bounding box getting stuck after a few frames. On the other hand, an increased sigma_position, during the initial horizontal motion, may result in a loss of tracking accuracy due to an overly diffused particle cloud. Yet, when bouncing occurs, the model struggles to adapt to the dynamic scene, losing proper tracking, especially in the presence of a dark area in the lower-right part of the video. This dark area introduces a unique challenge, preventing the model from executing proper tracking of the ball after rebounding, despite the theoretical expectations based on system noise adjustments.

- *What is the effect of assuming decreased/increased measurement noise?*

  A reduced sigma_observe, as evidenced in video3, seems to align with the expectations discussed earlier. Tracking generally performs well, but toward the final frames, a slight shift in the bounding box occurs. This shift could be attributed to particles becoming overly sensitive to color information in each frame, particularly in regions affected by lighting changes. Conversely, an increased sigma_observe, as seen in the video, results in a significant loss of tracking accuracy. The trajectory becomes independent of the object of interest, and the broader weight distribution of particles poses challenges in distinguishing between accurate representations of the ball's color histogram and inaccurate ones. Consequently, particles inaccurately representing the ball's position may be assigned disproportionately high weights, leading to a less reliable tracking outcome.

3.4 Final questions

- ***What is the effect of using more or fewer particles?***

  In the context of the three provided videos, the effect of using more or fewer particles becomes evident in shaping the tracking algorithm's performance.

  Video1, featuring a moving hand against a uniform background, benefits from a moderate number of particles, offering a good compromise between computational efficiency and accurate tracking.

  However, in Video2, where the hand encounters clutter and occlusions, an increased number of particles becomes advantageous. More particles allow the algorithm to better adapt to complex scenarios, improving robustness against occlusions and maintaining accurate tracking even in challenging conditions.

  On the other hand, Video3, depicting a bouncing ball, showcases the importance of a higher number of particles to capture abrupt changes in motion direction during bouncing.

  In summary, adjusting the number of particles is a critical parameter that needs to be fine-tuned based on the specific characteristics of the video, balancing computational efficiency with the algorithm's ability to handle dynamic scenarios, occlusions, and abrupt changes in motion direction.

- ***What is the effect of using more or fewer bins in the histogram color model?***

  Considering the three provided videos, the number of bins in the histogram color model plays a crucial role in the tracking algorithm's efficacy.

  In Video1, showcasing a moving hand against a uniform background, the presence of changing lighting conditions on the hand necessitates a higher number of bins. This adjustment enables the algorithm to capture subtle variations in color and effectively adapt to changes in illumination, ensuring robust tracking.

  Video2, featuring a moving hand with clutter and occlusions, benefits from an increased number of bins as it allows for a more detailed characterization of color distributions. This enhancement empowers the algorithm to discern between various objects and navigate occlusion challenges more adeptly.

  Similarly, Video3, illustrating a bouncing ball, benefits from a moderate number of bins to accurately represent the dynamic color variations during motion and bouncing events.

  Thus, the selection of the number of bins should be tailored to the specific video characteristics, addressing both the need for detailed color information and adaptability to changing conditions.

- ***What is the advantage/disadvantage of allowing appearance model updating?***

  In the context of the three provided videos, enabling appearance model updating introduces both advantages and disadvantages.

  For Video1, featuring a moving hand against a uniform background, updating the appearance model can be advantageous. It allows the algorithm to adapt to changes in lighting conditions, ensuring that the hand's color distribution is accurately represented throughout the video.

  In Video2, where a moving hand encounters clutter and occlusions, appearance model updating can be advantageous for adapting to variations in the hand's appearance caused by occlusions or interactions with other objects. However, the disadvantage is the risk of updating the model with irrelevant information from the clutter or occluded regions, leading to a less accurate representation of the hand.

  For Video3, showing a bouncing ball, appearance model updating might have limited advantages. The ball's appearance remains relatively consistent, and the bouncing events might not require frequent updates to the appearance model. However, unnecessary updates could introduce noise and potentially hinder tracking accuracy.

  In summary, the advantage of appearance model updating is its adaptability to changes in appearance, but the disadvantage is the potential inclusion of irrelevant information or sensitivity to noise, which must be carefully balanced based on the specific characteristics of the video.