Lab assignment 6

ETH Zurich, Computer Science Department

Computer vision

Report

Condensation tracker

Author
Emilia Szymańska, 22-945-547

December 2022

The objective of this lab assignment was to implement a CONDENSATION tracker based on color histograms and to run experiments to test the influence of the parameters for different types of videos on the performance.

1 CONDENSATION Tracker Based On Color Histograms

1.1 Color histogram

The first step was to calculate the normalized histogram of RGB colors within the defined bounding box. I slice the frame in accordance to the minimum and maximum coordinates provided as our bounding box description. Then I separate the RGB channels and for each of them I calculate the color histogram with the provided number of bins. Lastly, I normalize the histograms by diving them by their sums and create an output array containing histograms for three channels.

The code implementation of this part is presented below:

```
1  def color_histogram(xmin, ymin, xmax, ymax, frame, hist_bin):
2
3      sliced_img = frame[ymin:ymax,xmin:xmax]
4
5      red_channel   = sliced_img[:,:,0].flatten()
6      green_channel = sliced_img[:,:,1].flatten()
7      blue_channel  = sliced_img[:,:,2].flatten()
8
9      hist_red,   _ = np.histogram(red_channel, bins=hist_bin)
10     hist_green, _ = np.histogram(green_channel, bins=hist_bin)
11     hist_blue,  _ = np.histogram(blue_channel, bins=hist_bin)
12
13     hist_red   = hist_red/np.sum(hist_red)
14     hist_green = hist_green/np.sum(hist_green)
15     hist_blue  = hist_blue/np.sum(hist_blue)
16
17     hist = np.array([hist_red, hist_green, hist_blue])
18
19     return hist
```

1.2 Deriving A matrix

The state we consider is in form:

$$s = \{x, y, \dot{x}, \dot{y}\} \tag{1}$$

where x, y represent the center of the bounding box and $\dot{x}, \dot{y}$ are the velocities in x and y directions. Therefore, for no motion prediction model we do not want to change the values of the input vector, so A should be an identity matrix.

When we consider the constant velocity motion model, we want to add $\dot{x}, \dot{y}$ to x, y respectively. Therefore the matrix A is of form:

$$A = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2}$$

1.3 Propagation

This function propagates the particles given the system prediction model (matrix A) and the system model noise parameters in accordance to equation $s' = As + w$. First I define A matrix for both cases as well as the noise vector which is a normal distribution with the given sigma (varying for position and velocity values). New particles are computed by multiplying the matrix A with old particles and adding the noise vector. Finally, I make sure that the new particles lie within image dimensions.

The code implementation of this part is presented below:

```python
def propagate(particles, frame_height, frame_width, params):

    if params['model'] == 0: # no motion
        A = np.identity(2)
        w = np.transpose(np.random.normal(0, params['sigma_position'], size=
            particles.shape))
    else:                    # motion with const vel
        A = np.array([[1,0,1,0], [0,1,0,1], [0,0,1,0], [0,0,0,1]])
        w = np.transpose(np.random.normal(0, [params['sigma_position'], params['
            sigma_position'], params['sigma_velocity'], params['sigma_velocity']],
             size=particles.shape))

    new_particles = np.dot(A, np.transpose(particles)) + w
    new_particles = np.transpose(new_particles)

    new_particles = np.where(new_particles < 0, 0, new_particles)

    for part in new_particles:
        if part[0] > frame_width-1:
            part[0] = frame_width-1
        if part[1] > frame_height-1:
            part[1] = frame_height-1

    return new_particles
```

1.4 Observation

In the observe() function, for each particle I calculate the bounding box corners' minimum and maximum coordinates (and make sure they lie within the frame. Then I calculate the color histogram for this bounding box and later I compute the weight in accordance to the formula:

$$w = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{\chi^2(CH_s, CH_{target})^2}{2\sigma^2}} \tag{3}$$

I store all the weights in a list and once I compute them for all the particles, I create a numpy array from them and normalize them.

The code implementation of this part is presented below:

```python
def observe(particles, frame, bbox_height, bbox_width, params_hist_bin, hist,
    params_sigma_observe):
    height, width, _ = frame.shape

    norm_term = 1/(math.sqrt(2*math.pi)*params_sigma_observe)
    weights = []

    for part in particles:
        x_max = int(part[0]+bbox_width//2)
        x_min = int(part[0]-bbox_width//2)
        if x_max > width-1:
            x_max = width-1
        if x_min < 0:
            x_min = 0

        y_max = int(part[1]+bbox_height//2)
        y_min = int(part[1]-bbox_height//2)
        if y_max > height-1:
            y_max = height-1
        if y_min < 0:
            y_min = 0

        hist_observed = color_histogram(x_min, y_min, x_max, y_max, frame,
            params_hist_bin)
```

```
23
24          w = norm_term * math.exp(-(chi2_cost(hist_observed, hist)**2)/(2*
                params_sigma_observe**2))
25          weights.append(w)
26
27      weights = np.array(weights)
28      weights = weights/np.sum(weights)
29
30      return weights[:, None]
```

1.5 Estimation

In this step, I compute the estimate of the mean state, which is a sum of the product of particles and their weights.

The code implementation of this part is presented below:

```
1 def estimate(particles, particles_w):
2     return np.sum(particles_w*particles, axis=0)
```

1.6 Resampling

Finally, in accordance to the particles' weights, I randomly select new particles. The corresponding weights need to be normalized before being returned.

The code implementation of this part is presented below:

```
1 ...
2 def resample(particles, particles_w):
3     chosen_ids = np.random.choice(np.arange(particles.shape[0]), particles.shape
          [0], p=particles_w.flatten())
4
5     chosen_particles   = np.take(particles, chosen_ids, axis=0)
6     chosen_particles_w = np.take(particles_w, chosen_ids, axis=0)
7     chosen_particles_w = chosen_particles_w/np.sum(chosen_particles_w)
8
9     return chosen_particles, chosen_particles_w
10 ...
```

2 Experiments

2.1 Video 1

The first video runs with the tracker as presented in Fig. 1 for no motion model and Fig. 2 for constant velocity model. It can be seen that the tracker manages to follow the top part of the hand, which was selected for tracking, however it was more noisy for the constant velocity model. The blue trajectory corresponds to a priori mean state and the red trajectory to a posteriori mean state.

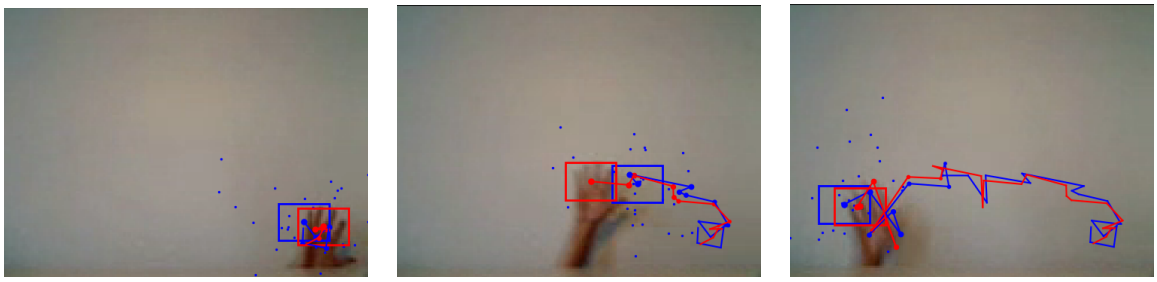


Figure 1: Result of running the tracker with no motion model on the video nr 1 (tracking the hand).

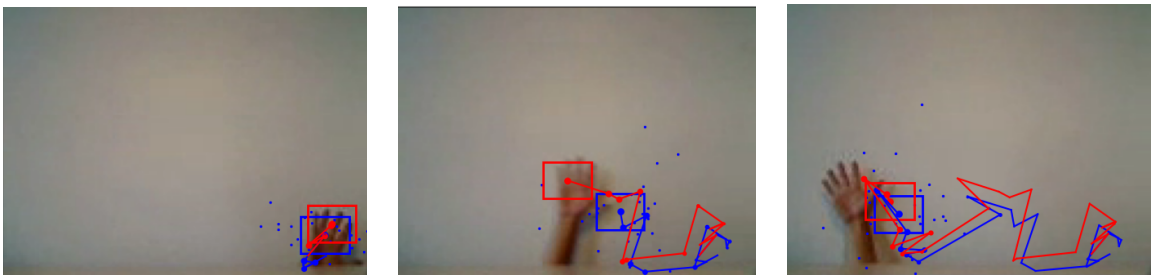


Figure 2: Result of running the tracker with constant velocity model on the video nr 1 (tracking the hand).

2.2 Video 2

The second video's objective is to track a hand, which at some point is occluded by another object. This time I was supposed to vary the parameters to obtain the final tracking (Fig. 3). I run the video for different σ values and modes of operation (Fig. 5, 4). The tracking had better results for constant velocity model – for no motion model the correct tracking was usually stopped when encountering the occluding obstacle. Increasing $\sigma_{position}$ resulted in highly "scattering" behaviour, but decreasing it made the tracker prefer states without almost any movement. With high $\sigma_{velocity}$, the tracker performed big steps between frames, the opposite happening for low $\sigma_{velocity}$ values. As for measurement noise, using a high $\sigma_{observe}$ resulted in a priori and a posteriori trajectories to be almost exactly the same, therefore not following the object. Significantly decreasing $\sigma_{observe}$, however, will also not be a solution - the trajectories are also chaotic (although more accurate than in the other case).

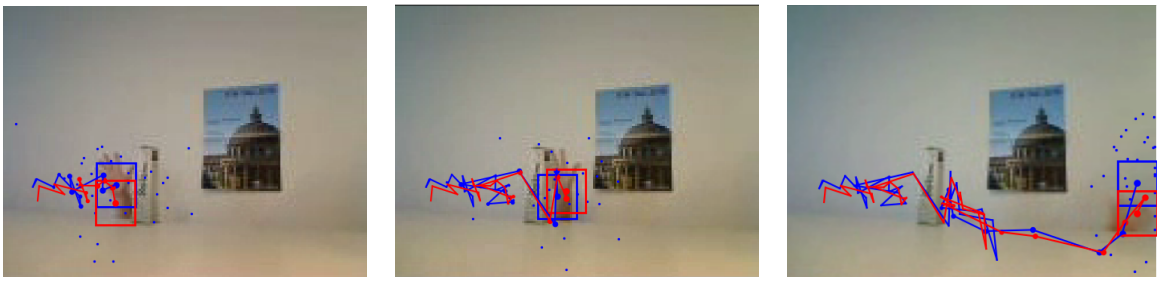
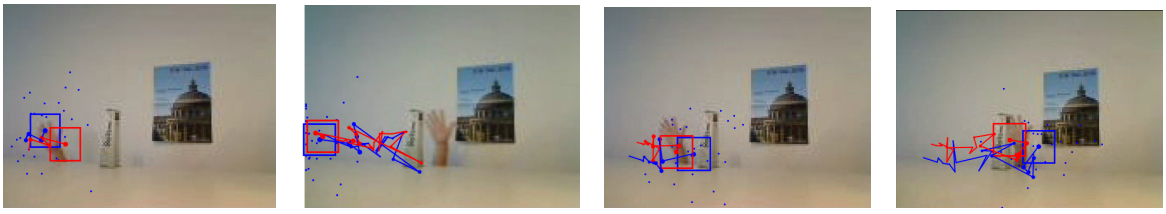
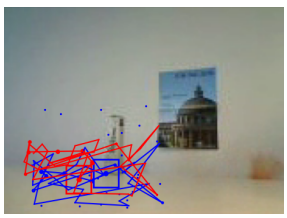


Figure 3: Tracking the hand on the video nr 2.



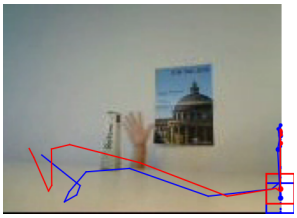
(a) No motion (b) No motion (c) Constant velocity (d) Constant velocity

Figure 4: Comparison of no motion and constant velocity model for the tracker on video nr 2.

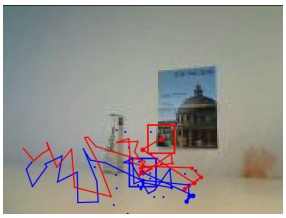


(a) High $\sigma_{position}$

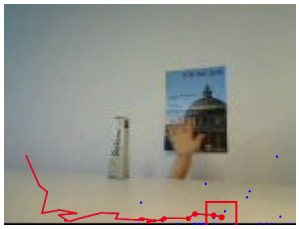
(b) Low $\sigma_{position}$



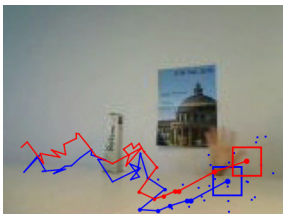
(c) High $\sigma_{velocity}$



(d) Low $\sigma_{velocity}$



(e) High $\sigma_{observe}$



(f) Low $\sigma_{observe}$

Figure 5: Comparison of σ magnitudes for velocity, position and observation for video nr 2.

2.3 Video 3

In my case, plugging the parameters from the video nr 2 to the video nr 3 did not allow for tracking the ball (Fig. 6) – the tracker got "stuck" after the ball changed its direction of movement. I needed to tune the parameters to make it possible to properly follow the ball's movement (Fig. 7). The comparison for $\sigma_{position}$ values for this video yields similar results to the $\sigma_{position}$ varying of video nr 2 – it affects the "willingness" to change the position of the center of the bounding box. It is however slightly harder to spot the difference here when varying $\sigma_{velocity}$ – we can see that with a lower value it is more likely to be "stuck" in the very corner of the image. High $\sigma_{observe}$ value again makes the a priori and a posteriori trajectories to be equal and they do not move fully to the direction of the ball movement (they however managed in this case to partially "find" the ball again when it started coming back to the initial position), while decreasing it results in more noisy behaviour of both trajectories. The comparison of σ values and their effects on the videos is preented in Fig. 8. Considering changing the model from no motion to constant velocity one, there is a significant difference in the result. The good trajectory in Fig. 7 was based on the no motion model and constant velocity one resulted in similar behaviour to Fig. 6 – the tracker stopped tracking when the ball hit the wall and changed the movement.

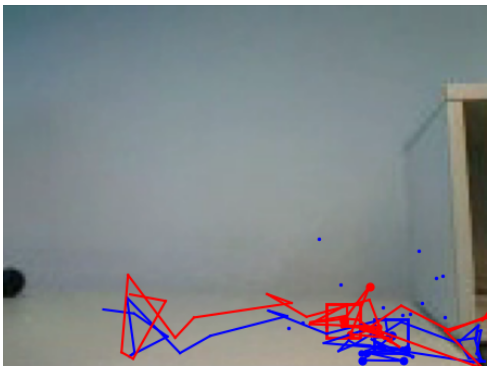


Figure 6: The effect of using the parameters tuned for video 2 on video 3.

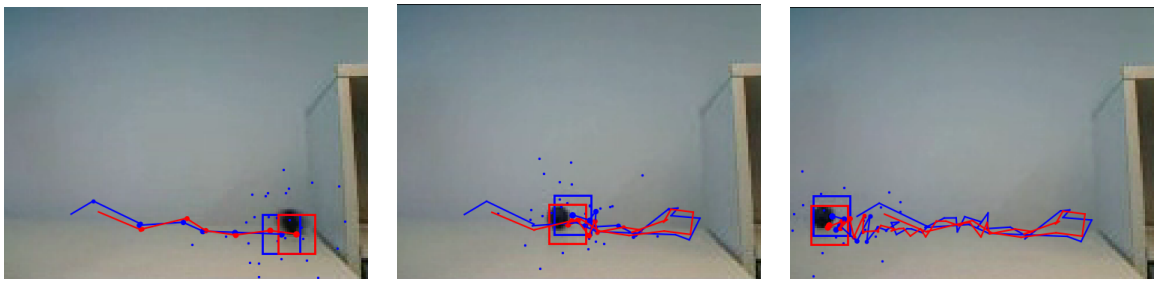
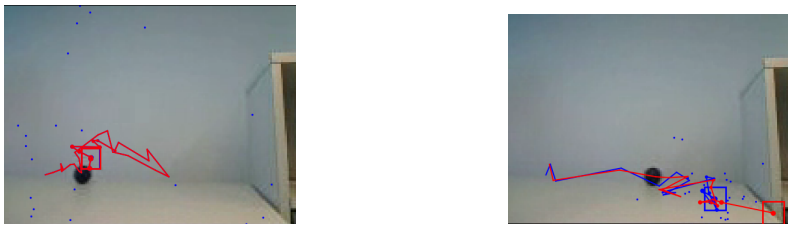


Figure 7: Tracking the ball on video 3.

(a) High $\sigma_{position}$ (b) Low $\sigma_{position}$ (c) High $\sigma_{velocity}$ (d) Low $\sigma_{velocity}$



(e) High $\sigma_{observe}$ (f) Low $\sigma_{observe}$

Figure 8: Comparison of σ magnitudes for velocity, position and observation for video nr 3.

2.4 General observations

In all the videos, there were parameters which had similar results when being changed. Having fewer bins in histogram color model made the tracking more general, therefore it was less accurate (we made the disctinction on fewer features). If we increase this number, we will obtain better results, however at some point the tracker can become too specific when looking for similarities and it may not be accurate anymore. Allowing the appearance model to be updated after each iteration adjusts it to the current surrounding, which in a varying environment might be useful (in our case we mostly had plain wall throughout the video, but if on the way the background is significantly different than the starting one, we might lose the correct tracking). The number of particles affects the choice of the center of the tracking bounding box – again, usually more is better for tracking the object, however too many may result in too specific search and taking into consideration data which is not relevant. If we do not have enough particles, the result will be highly noisy because of the lack of informative data.