Tracking - laboratory

Federico Magnolfi

30 May 2019

Before starting

Ensure you have these libraries:

- opencv-python 4.1.0
- opencv-contrib-python 4.1.0
- imutils
- matplotlib

Recap: what is tracking

Object tracking is the process of:

- Taking an initial set of object detections (a set of bounding boxes)
- Creating a unique ID for each of the initial detections
- Tracking each of the objects as they move across frames in a video, maintaining the assignment of unique IDs

Recap: why use tracking

Tracking can be useful because:

- Makes easy to assign IDs to objects
- Can be much faster than object detection, this is ideal for real-time contexts and when there are many objects to track
- Can help when detection fails

MOSSE vs KCF vs CSRT

As tracking algorithms we will use:

- MOSSE (2010)
- KCF (2014)
- CSRT (2017)

From the theory of these three algorithms:
$\downarrow$ better quality of tracking
$\uparrow$ faster

What we want to do

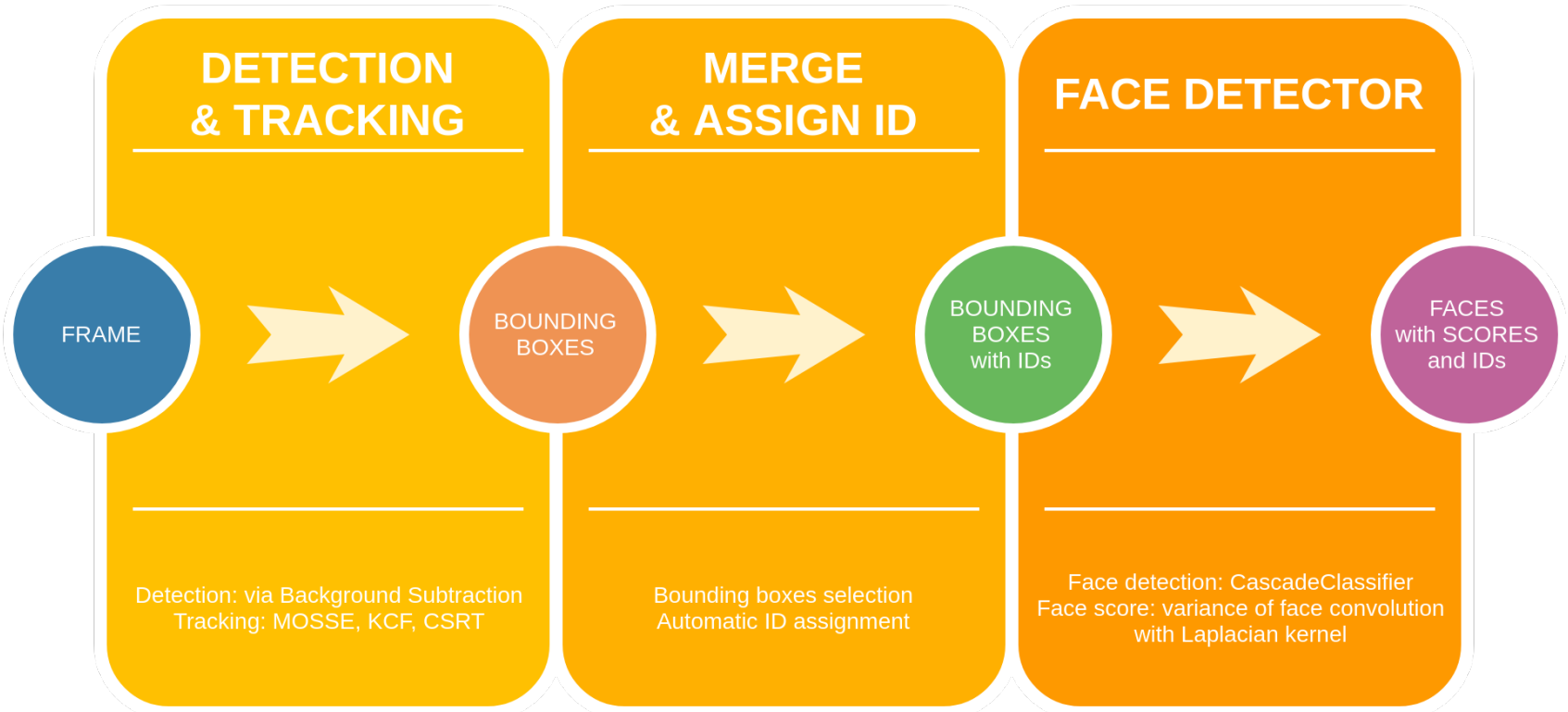
Description

We have a video camera and potentially many people could pass by it.

Objective

We want to register the (best) face(s) of each person.

How to reach the objective



Background subtraction in OpenCV

There are many BG subtractor algorithms in OpenCV, example creation of some background subtractor (with their default parameters):

```
bM = cv2.bgsegm.createBackgroundSubtractorMOG(history=200,
      nmixtures=5, backgroundRatio=0.7, noiseSigma=0)
bM2 = cv2.createBackgroundSubtractorMOG2(history=500,
      varThreshold=16, detectShadows=True)
bG = cv2.bgsegm.createBackgroundSubtractorGMG(
      initializationFrames=120, decisionThreshold=0.8)
bK = cv2.createBackgroundSubtractorKNN(history=500,
      dist2Threshold=400.0, detectShadows=True)
```

Example usage of a background subtractor:

```
fgmask = myBgSubtractor.apply(frame)
```

Tracking in OpenCV

There are different trackers in OpenCV, each one with the same interface. We pick CSRT as example but the other are analogous.

Tracker creation

```
tracker = cv2.TrackerCSRT_create()
```

Tracker initialization

```
tracker.init(frame, boundingBox)
# boundingBox as (left, top, width, height)
```

Tracker locate and update

```
(success, boundingBox) = trackers.update(frame)
# success is False if tracking has failed
# boundingBox is the location of the tracked object
            in the frame
```

In Python we can't specify parameters in *create*, while we can in C++

MultiTracking in OpenCV

OpenCV has a *cv2.MultiTracker* class that can handle many Trackers at the same time.

cv2.MultiTracker issue (at least in OpenCV 4.1.0)

cv2.MultiTracker update method doesn't return success for each specific tracker: success is False if **at least one** tracker has failed. Therefore, it's impossible to differentiate failure management between trackers.

To avoid *cv2.MultiTracker* issue, we implement a class with the same interface, but that returns a list of successes instead of only a boolean success.

Implemented classes

- *CompositeBackgroundSubtractor*: it allows to use more background subtractors simultaneously, and use as a segmentation mask the mask obtained by the OR of the individual masks
- *ProcessPipeline*: makes it easy to specify a series of functions to be applied to segmentation masks in order to remove noise and make objects better recognizable
- *ObjectDetector*: uses a background subtractor and a pipeline to detect objects in frames
- *FaceDetector*: recognizes faces and assigns scores to them (a high score indicates good picture quality)

Implemented classes

- *Tracker*: extends *cv2.Tracker* by aggregation, adding information as ID and number of failures, while maintaining the same interface of *cv2.Tracker*
  (i.e. has the same methods *init* and *update*)
- *TrackerManager*: takes the place of *cv2.MultiTracker* taking charge of managing all the trackers
  (i.e. trackers creation, delete, merging...)

In one of the exercises, we'll implement some methods of *TrackerManager*.

How to properly take advantage of tracking?

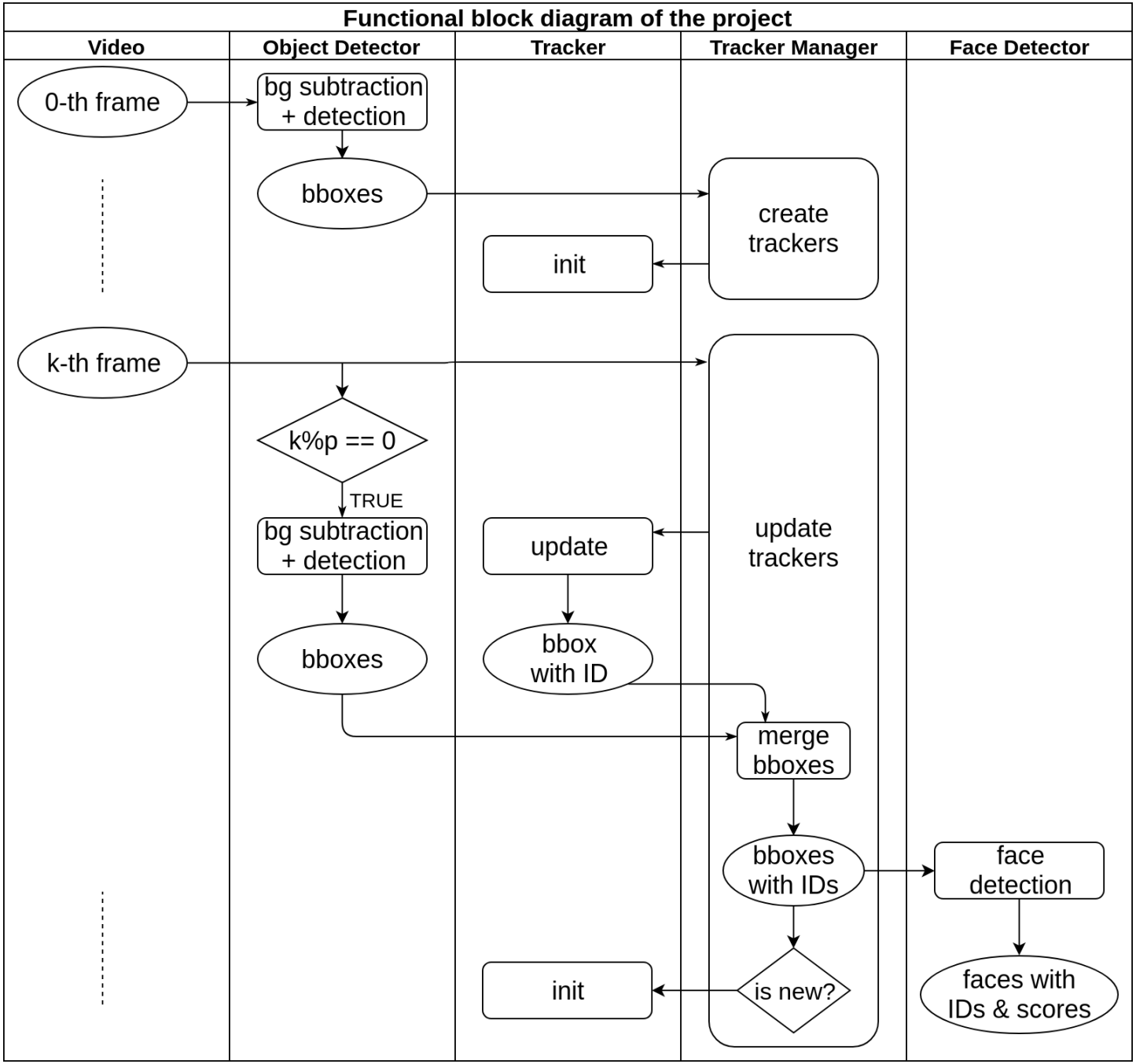
The answer is not obvious: there are different possibilities.
Questions:

- When to do detection? $\rightarrow$ Periodically
- When to do tracking? $\rightarrow$ Potentially every frame
- It makes sense to do both detection and tracking in the same frame? $\rightarrow$ Yes
- If so, how to treat the detection and tracking outputs? How to merge them? $\rightarrow$ Let's try different solutions

Functional block diagram of the project

In the next slide there will be the functional block diagram of the project, here is a legend for some abbreviations that appear in the diagram:

- *bg subtraction*: background subtraction
- *p*: period length, in frames (i.e., every how many frames we make the object detection task)
- *bbox(es)*: bounding box(es)


Functional block diagram of the project
Video
Object Detector
Tracker
Tracker Manager
Face Detector
0-th frame
bg subtraction + detection
bboxes
create trackers
init
k-th frame
k%p == 0
TRUE
bg subtraction + detection
update
update trackers
bboxes
bbox with ID
merge bboxes
bboxes with IDs
face detection
init
is new?
faces with IDs & scores

Exercises

Dataset

Our dataset consists of 3 videos:

- *video_116.mp4*: the simplest video of the dataset
- *video_white.mp4*: a challenging video for the background subtraction task (foreground and background have a similar color)
- *video_205.mp4*: a challenging video for the tracking task (two people in the video, risk of identity exchange)

Dataset: example

video_116.mp4

video_white.mp4

video_205.mp4

Exercise 1

Motivation

Before starting using tracking, first of all we have to prepare a good detector: without a good detector, tracking is useless.

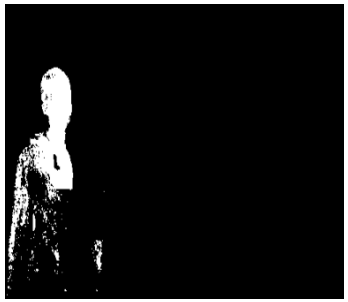
Objective

Create a good enough detector, using the background subtraction.

Background subtraction only is not satisfying: we have to further process the mask to remove the noise; only after noise removal we can put bounding boxes on foreground contours.

Exercise 1

Initial mask

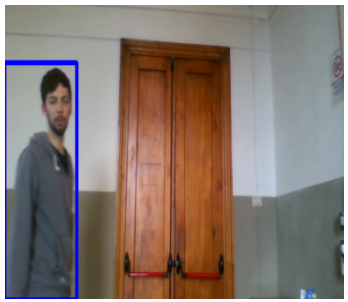


↓

Filtered mask

↓

Detection



- Different background subtractors give different segmentation masks
- We have to prepare the mask before object detection

Exercise 1: assignment

Let's prepare the detector

- **TODO 0**: select an input stream (video file or webcam: if you use a video, comment the line that flips the frame)
- **TODO 1**: fill the list with some background subtractor to try
- **TODO 2**: try to add functions to the pipeline, in order remove noise to the mask obtained by background subtraction (e.g.: *cv2.medianBlur*, *cv2.dilate*, *fillHoles*...)
- **TODO 3**: choose (once and for all) the width to resize the frame (remark: the height is automatically changed to keep the aspect ratio unchanged)
- **TODO 4**: choose a good pipeline and, for each video, choose best background subtractor algorithm and parameters

Exercise 1: what to check

- Are all the objects detected in almost every frame?
- Are the objects fully recognized? (We want only one rectangle per object, we don't want so many small rectangles)
- Are there few false positives?

Exercise 1: what to expect

Exercise 1: considerations

- Background subtractors adapt differently to the various videos
- Background subtractors change a lot depending on the parameters

Exercise 2

Motivation

To be able to use tracking, we must master the OpenCV Tracking API. Furthermore, we need to know how to overcome the deficiencies of *cv2.MultiTracker* class.

Objective

Our *TrackerManager* class takes the place of *cv2.MultiTracker*. The objective of this exercise is to implement two methods of this class.

In particular, we implement methods responsible of add/init a new tracker and remove all those who have failed too many times. After that, we'll discuss the implementation of the *update* method of the *TrackerManager*.

Exercise 2: some details of *Tracker*

Our *Tracker* class **constructor** takes two arguments:

- *tracker*: instance of *cv2.Tracker*
- *id*: identifier of the tracker. If None, an ID will be automatically assigned, otherwise will be used the specified ID

Remind

To instantiate a *cv2.Tracker*, we can use:

- `cv2.TrackerMOSSE_create()`
- `cv2.TrackerKCF_create()`
- `cv2.TrackerCSRT_create()`

Our *Tracker* class has many members, but you'll need:

- *id*: identifier of the tracker
- *numFailures*: current number of failures (after the last successful localization)

Exercise 2: some details of *TrackerManager*

It has three members:

- *trackers*: list of all trackers managed (each tracker must be an instance of our class *Tracker*)
- *nameDefaultTracker*: name of the default Tracker class to create trackers ("MOSSE" or "KCF" or "CSRT")
- *maxFailures*: maximum number of consecutive frames in which the tracker can fail

Exercise 2: some details of *TrackerManager*

It has many methods, but when we'll discuss *update* we'll use:

- `mergeBBoxes(successes,bboxes,detObjs,IDs,maintainDet)`
  Merge trackers' and detector's bounding boxes, resolving the conflicts (overlaps)
- `reinitTracker(self, objID, frame, obj_bbox)`
  Create and initialize a new tracker that replaces an existing one, while maintaining the same identifier (this is necessary when we want to force the change of object bounding box)

Exercise 2: assignment

- **TODO 5,6**: implement method

  `addTracker(self, frame, obj_bbox, trackerName=None)`

  Create a tracker depending on the specified name (pick default name if *trackerName* is None). Then initialize the tracker and accordingly update the fields of the class
- **TODO 7**: implement method

  `removeDeadTrackers(self)`

  Remove all trackers that has exceeded the number of maximum allowed failures

Deepening on *TrackerManager*

Managing *update* consists in update all trackers on the given frame, and eventually merge tracking bounding boxes with detection ones. Furthermore:

- A new tracker must be created for each new object
- If we want to keep detection bounding boxes when conflicts arise during merge, we have to reinitialize involved trackers

Let's have a look at the implementation of *update* method, using *mergeBBoxes* and *reinitTracker* as black-box functions.

Deepening on *TrackerManager*: *update* method

```
"""
frame: the frame where to search for the objects
detObjs: list of bounding boxes given by the object detector
maintDet: True to maintain detector's bboxes in case of overlaps
return: the list of successes (Boolean) and the list of bounding boxes (x,y,w,h)
"""
def update(self, frame, detObjs=None, maintDet=True):
    successes,bboxes=self._update(frame) # update all trackers, without merge
    if detObjs is not None and detObjs != []: # if the merge has to be done
        trkIDs = self.getIDs() # get the IDs of tracked objects
        # now let's merge detector's and trackers' bboxes
        successes, bboxes, trkIDs, changes = self.mergeBBoxes(
              successes, bboxes, detObjs, trkIDs=trkIDs, maintainDetected=maintDet)
# merge returns 4 lists in which the same index refers to the same object:
# successes[i]: True if tracker of index i has successfully located the target
# bboxes[i]: bounding box (x,y,w,h) of object i-th
# objIDs[i]: ID of the object; it is >= 0 if existed previously, otherwise -1
# changes[i]:True if i-th object was detected or its bbox changed(in this frame)

        for bbox, objID, change in zip(bboxes, objIDs, changes):
            if change: # True if object is new or its bbox has changed in merge
                if objID == -1: # this object does not have an ID yet
                    self.addTracker(frame, bbox)
                else: # this object already has an ID
                    self.reinitTracker(objID, frame, bbox)

    return successes, bboxes
```

Exercise 3

Motivation

Finally, we must try to put together tracking and detection.

Objective

Combine tracking and detection to get the best possible result.

Exercise 3: assignment

- **TODO 8,10,12,13**: complete with values obtained from exercise 1
- **TODO 9**: for each video, change the tracker and check the results
- **TODO 11**: for each video, change the values of *period* and *maintainDetected*, and check the results

Exercise 3: what to check

- Do rectangles follow objects well?
- The ID assigned to an object always remains the same?
- Does the drifting phenomenon occur?
- How fast is video analysis?

Exercise 3: what to expect

Exercise 3: considerations

- If period is too long, detection fails (the videos are short and therefore the frames seen by the subtractor become few)
- Find a good balance between tracking quality and speed is difficult
- In transition frames it is preferable to keep the bounding boxes of the detector
- Theory is confirmed: CSRT is the most reliable tracker, while MOSSE is the fastest one (about the ones we tried)

Final considerations

- With tracking we can easily assign **IDs** to objects that appear in a video
- A tracker **must** rely on a good detector
- To better use tracking in a **real-case** scenario, we must analyze the specific case: which is the resolution of the camera? Which is the typical noise introduced by the environment? How do we expect the objects to be traced to appear? How many computational resource we have?