Multi-Object Tracking

Outline

1. Introduction

2. Task Description

3. Concepts
    a. Object Tracking
    b. Benefits of Using Trackers
    c. Tracking Algorithms

4. Samples

5. Possible Challenges
    a. How to track multiple objects simultaneously?
    b. How to generate unique IDs for every tracked-object?
    c. How to preserve this unique ID?

6. Design & Workflow
    a. Software Structure
    b. Implementation

7. Results & Conclusion

8. References

Introduction

Artificial Intelligence has many real-life applications, and almost everyone, though not consciously, uses these applications and their benefits. It is used in many fields of studies other than Computer Science, from Neuroscience to Agriculture. Therefore, it is essential to study AI and other related fields, such as Machine Learning, Deep Learning, and Computer Vision.

Due to the enormous amount of data available thanks to the Internet, Machine Learning and Computer Vision have been growing at a fast pace throughout the previous decade. This growth is going to be even more rapid than before. So, there are numerous applications, therefore, tons of profit and revenue in these fields.

Computer Vision is now being used everywhere, in our cellphones, in social media, security systems, self-driving cars, industries, and even medical sciences. One of the integral parts of Computer-Vision-based systems is Object Detection & Tracking. Although they are not very new concepts, their substantial scientific growth began in the 2000s. Since then, many new ideas have been introduced, and many different algorithms have been developed and implemented by a large community.

As mentioned above, one of the applications of Object Detection and Tracking is in the security systems. Advanced security systems need to detect the objects (pedestrians, for instance) and track them to identify and categorize their actions, and then act accordingly. In this project, I will develop an application using OpenCV and C++ that works as the foundation of these systems.

Task Description

In this task, the application should detect the objects in a video using a CNN-based object-detector, then assign a unique ID to each of them and track them, until they are present in the frames, or get occluded by other objects. Additionally, every tracked object should maintain its ID throughout its presence in the scene. ID preservation is essential to keep track of an object since different parts of the comprehensive system require it to perform their task. It is also necessary that the program runs in real-time and have a high FPS because of the nature of its application. This application's environment is live, so the system needs to do its job very quickly and with minimum delay.

Concepts

Object Tracking

An object tracker is needed to track the objects of the scene. Various ideas and algorithms are in use, each having pros and cons and specific applications. The algorithm we select for a given task depends on the environment in which the program is going to be used. By comparing some of these algorithms and their features, which I am going to explain in the following, I decided to use the KCF tracker as the default for this project; however, thanks to OpenCV Tracking API, it is possible to switch between algorithms without much effort.

Benefits of Using Trackers

1. **They are fast:** Executing an object-detection algorithm is computationally expensive and time-consuming. In contrast, modern object trackers are much quicker than detectors because the object-detection algorithms require much more computation. Also, object-detectors make predictions based on the whole frames, but trackers only need to care about a fraction of a frame, often called the Region of Interest (ROI).

2. **They could be more reliable:** Sometimes, the object-detector cannot detect an object consistently in a sequence of frames; one possible reason is that the model is not trained to detect the object in which we are interested. Using a tracker, even if the detector performs poorly and just detects the object in a portion of frames, it is possible to handle this situation.

3. **They can preserve identity:** Object detectors do not keep any memory of their previous actions and states; therefore, each frame they process is entirely a new input for them. They do not care if the input is a video's frame sequence or an array of arbitrary images. On the other hand, trackers keep track of their ROI, and they could tell if their ROI in a new frame corresponds to their previous ROI or not.

Tracking Algorithms

In this section, some trackers will be introduced very briefly.

1. **Boosting:** This tracker is one of the oldest trackers, and there is no reason to use it since other trackers will perform faster and with more accuracy.
2. **MIL**: It is a kind of an extension of the Boosting tracker but more accurate. However, its computation cost is high.
3. **KCF**: This tracker executes faster, and also its accuracy is acceptable. One major advantage of this tracker is that it reports tracking failures accurately and knows when an object is not present anymore.
4. **CSRT**: Very accurate, but also very time-consuming. It would be an excellent choice in offline environments, or for single-object tracking.

Possible Challenges

How to track multiple objects simultaneously?

- **Possible solution**: Define a class with a member vector of trackers, which updates all of the trackers present in the vector every time its update method is called.

How to generate unique IDs for every tracked-object?

- **Possible solution**: Define a static member variable for the trackers class and increment it each time a new object is detected.

How to preserve this unique ID?

- After each detection, it is necessary to match every tracker's ROI with its pair in the previous frame in order to maintain the ID.
- **Possible solution:** For each new ROI, among all previous ROIs find the nearest, and then calculate their Intersection Over Union (IOU). If this value is higher than a threshold, then we consider two ROIs as the same object. Also, for better results, we could apply feature matching techniques like SIFT-RANSAC to those boxes.

Design & Workflow

Software Structure

- A **main.cpp** that opens the video file and initializes the needed variables. It should also create the required object instances for detection and tracking, and initialize them properly using their constructors. The key part of **main.cpp** is the while-loop. This loop reads every frame in the input video, applies the detector to the frame (with X frames between each detection), sets the needed trackers, updates the trackers, and eventually shows the output on a **cv::namedWindow**.

- A **MyTrackers** class which is responsible for creating and updating every single tracker and assigning them an ID. Using an object of **MyTrackers** class, the main function could easily receive the updated bounding boxes with no knowledge of trackers' function.
    - This class should have two public methods for two important tasks.
        1. Creating and initializing single trackers for every new bounding box → **.setTrackers(&frame, &boundingBoxes)**
        2. Updating each tracker with new frames → **.update(&frame)**

- A **MyTracker** class whose objects contain a **cv::Tracker**, an ROI, and an ID. The class objects are responsible for keeping the data associated with every single **cv:: Tracker**.

- A **MyDetector** class. Like **MyTrackers** class, only one object of this class is needed in the code. It will be responsible for detecting the objects in the frames, and only have one public method: **.detect(&frame)**. This method will be responsible for returning the bounding boxes surrounding each object in the given frame, which will be the input of the **trackers.setTrackers()**. Now, this class only supports darknet and uses yolov3 as default.

Implementation

Following the suggested steps, I implemented the program in these steps:

1. Using **cv::selectROI()**, I initialized a single **cv::Tracker** and then updated it in all the following frames. At this step, I tried different tracker types and found out which one will be more appropriate for this task.
2. Using **cv::selectROIs()**, I drew multiple ROIs by hand and fed them to a **cv::MultiTracker** object to initialize it. Then, for all other frames, I called the **.update()** method of this object to track multiple ROIs simultaneously.
3. I realized that **cv::MultiTracker** would not be a fit for my needs because of its limitations, so I decided not to use it for handling multiple trackers and created my own **MyTrackers** class.
4. I implemented the Detection class. Using its **.detect()** output, I initialized the trackers instead of using **cv::selectROI** or **cv::selecROI**.
5. Added a static member variable to **MyTrackers** class named **nextTrackerId**. This variable's value will be passed to **MyTracker** setter function and increment after a new box insertion; therefore, the IDs will be unique for each tracked-object. After that, I implemented the box matching algorithm. This algorithm will run after each detection step (in **trackers.setTrackers() method**) and tries to find a match for each new box in the previous boxes set. If a match is found, a new tracker for this box will be initialized using its pair's ID. Otherwise, the new box will use a new ID, and the **nextTrackerId** will increment.
6. Added an argument parser to the software to make it easier to change code input values such as threshold values. Using the argument parser, it is quite easy to test the effect of changing different input values. It can also set detection-step, change the tracking algorithm, change the input video and model, and enable fullscreen mode.

Conclusion & Future Work

After testing the application several times on different videos, I adjusted the threshold values to achieve the best possible result. It worked quite well, and I was almost satisfied with the result. It is essential to optimize the code for the real environment in which it will be used. No best optimization exists for all the cases and scenarios. Apart from that, general improvements could be applied, of course. Here is a list that I came up with for improving the application:

1. Even if the detector fails to detect the tracked-object, try to track it using the previous ROI.
2. Make the box-matching algorithm efficient.
3. When a tracked-object is lost (and not found after a while), make its ID available to newly added objects. (I did this using a timer, for now, making ID counter be set to 0 after a particular time interval)
4. Make the trackers more efficient and let them run in parallel.
5. Add support for various object detectors.
6. Add more functions to GUI.
7. Make the NMS more robust.
8. Ultimately run the code on GPU.

References

1. https://docs.opencv.org/
2. www.learnopencv.com
3. www.pyimagesearch.com