Nizhny Novgorod State University
Institute of Information Technologies, Mathematics and Mechanics
Department of Computer Software and Supercomputer Technologies

Educational course «Modern methods and technologies of deep learning in computer vision»

Practice №3
Video object tracking

Supported by Intel

Vasiliev E. P.

Nizhny Novgorod
2020

Content

1 Introduction

In this practice, we solve the problem of object tracking via constructing object movement trajectories and develop a solution based on the Intel Distribution of OpenVINO Toolkit. Pre-trained deep models from the Open Model Zoo repository are used [5].

2 Guidelines

2.1 Goals and tasks

The goal of this practice is to study the problem of object tracking and implement an algorithm that solves the problem via matching detections (tracking-by-detection approach), using the Intel Distribution of OpenVINO Toolkit.

To achieve this goal, it is required to solve the following tasks:

- Study the algorithm of object tracking based on matching approach and implement it in Python.
- Develop an application based on the Inference Engine for object tracking. The tracking result should be displayed on the original video.
- Execute and verify the developed sample.

2.2 Practice structure

First, a brief description of object tracking via matching detections is provided. Further, the application for solving the problem of object tracking is developed step-by-step.

2.3 Recommended study sequence

The recommended practice sequence is as follows:

- Study tracking-by-detection approach and matching detection algorithm.
- Setup software environment for using the Intel Distribution of OpenVINO Toolkit.
- Develop an application based on the Inference Engine for object tracking. The tracking result should be displayed on the original video.

Note that the environment setup is described in detail in the first practice; so this step is omitted in this tutorial.

3 Object tracking via matching detections

3.1 General scheme of tracking algorithm

An input of object tracking is a sequence of video frames. An output is a set of object location sequences on the input frames. The formal problem statement is represented in the lecture “Deep models for tracking objects in the video” of this course.

The general scheme of the algorithm consists of several stages. It is also assumed that at first step, separate parts of the tracks have already been constructed for a set of objects detected by the current step. A ***track*** is a sequence of object locations (bounding boxes) on video frames.

1. Extract current video frame.
2. Detect objects in this frame.
3. Calculate the similarity matrix between the detected objects and objects for which separate parts of the tracks are constructed.
4. Using the given similarity matrix, answer next questions:
   4.1. What tracks does the detected object correspond to?
   4.2. What objects appeared for the first time on the video (do not correspond to any of the existing tracks)?
   4.3. For which tracks on the new frame was the object not detected (the track ended because of the object leaving the camera's field of view)?

5. In accordance with the answers, update the positions in the tracks, create new tracks for newly discovered objects.
6. Repeat from the first step until the video ends.

The first step of the described scheme is performed by standard libraries. The second step supposes solving the problem of object detection using deep learning models. Object detection based on deep learning approach was described in the previous practice. Steps 3 – 5 are described in detail below.

3.2 Calculating similarity matrix

Similarity matrix A between N tracks and M objects is a matrix of the size $N \times M$, where each element a_{ij} represents the similarity coefficient between the track T_i and the object R_j. Using this matrix, you can find the best matches between the detected objects and existing tracks.

The easiest way to calculate the similarity coefficient of the track T_i and the new object R_j is shown below:

1. Extract the last object position (bounding box) from the track T_i.
2. Compare the object in the track and the detected object R_j according to some features.

To compare it is possible to use one or more of the following features: location, shape and appearance of the object. Let D is the distance between the centers of the bounding boxes (diagonals intersection), (w_1, h_1) is the width and height of the last bounding box in the track, (w_2, h_2) is the width and height of the detected bounding box, C_1, C_2 are the weights of the features into the similarity coefficient. Then the formula for calculating the similarity coefficient for the location feature is as follows:

$$affinity\_place = e^{-C_1\left(\frac{D^2}{w_1 h_1}\right)},$$

and the formula for calculating the similarity coefficient by the shape feature is represented below:

$$affinity\_shapes = e^{-C_2\left(\frac{w_1 - w_2}{w_1} + \frac{h_1 - h_2}{h_1}\right)}.$$

The similarity coefficient is determined as follows:

$$a_{ij} = affinity\_place * affinity\_shapes.$$

3.3 Searching for the best matches by similarity matrix

The problem of searching for the matches by the similarity matrix reduces to the assignment problem [3, 4, 6].

1. There are $\{1, \ldots, N\}$ agents and $\{1, \ldots, N\}$ tasks, which can be distributed between these agents.
2. Only one task can be assigned to each agent, and each task can be assigned to only one agent $j = f(i)$ with cost $a(i, j) \geq 0$.
3. The assignment problem is to find a feasible set of assignments $A = \{(i_1, j_1), \ldots, (i_n, j_n)\}$ of the minimum total cost: $\sum_j a\big(i, f(i)\big) \rightarrow min$.

The problem of searching for the best matches of the tracks and detections by the similarity matrix is the problem of maximizing the total similarity. In order to reduce this problem to the assignment problem, it is required to perform the following actions:

1. Make matrix A square. We are able to add a number of empty rows and columns filled by zeros.
2. Reduce the maximization problem to the minimization problem. Since $0 \leq a_{ij} \leq 1$, it is enough to replace each element in the similarity matrix by the formula $a'_{ij} = 1 - a_{ij}$.

To solve the assignment problem, you can use the **`linear_sum_assignment`** function of the **`scipy.optimize`** package, which implements the Hungarian algorithm [3, 4].

3.4 Updating tracks

When the matches of tracks and objects were constructed, it is required to update tracks.

1. If the similarity coefficient between the object and the track exceeds a threshold (usually in the range 0.02 – 0.2), then append the object to the track.
2. If the object is not appended to any track, then create a new track and append the detected object to this track.

4 Developing the object tracking application using the OpenVINO Toolkit

4.1 Installing additional dependencies

In this practice, we need to use the **scientific-python** library, so we should install the **SciPy** package. **SciPy** is a free library used for scientific and engineering calculations.

```
pip install scipy
```

4.2 File structure

For this practice, please, create two files**:** **matching_tracker.py** is a file containing classes **Tracklet** and **MatchingTracker**, and **tracking_sample.py** is a file containing the testing code for the implemented tracking algorithm.

The base entity, which contains information about the object location in the image, is the named tuple **DetectedObject**.

```
DetectedObject = namedtuple('DetectedObject',
   ['confidence', 'frame_idx', 'object_id', 'timestamp', 'class_id',
    'x_left', 'y_bottom', 'x_right', 'y_top',])
```

- **confidence** is a confidence of the object location in the selected area (float).
- **frame_idx** is a frame number (integer).
- **object_id** is a track identifier (integer, takes value -1 if the object is not assigned to any track).
- **timestamp** is a timestamp (integer, in milliseconds), it is required to track the point in time when an object was detected.
- **class_id** is an object class identifier (integer, used to display the class name on the screen).
- **x_left**, **y_bottom**, **x_right**, **y_top** are coordinates of the bounding boxes in the range from 0 to 1.

A class to represent a track is **Tracklet**. This class stores a list of **DetectedObject** objects and provides several methods.

- **__init__** is a constructor, it creates inside itself a list of **DetectedObject** for storing.
- **__len__** is a method to get the length of an object list.
- **__getitem__** is a method to receive an object by its number.
- **add_new_detection** is a method to add a new **DetectedObject** to the list.

```
class Tracklet():
    def __init__(self, detection=[]):
        self._trackedObjects = []
        self._trackedObjects.append(detection)
    def __len__(self):
        return len(self._trackedObjects)
    def __getitem__(self, position):
        return self._trackedObjects[position]
    def add_new_detection(self, detection):
        self._trackedObjects.append(detection)
```

There is a common practice in Python to create a new data type from named tuple and a class for storing and access to elements. To work with tracks in the paradigm of the Python language, the standard methods **__len__** and **__getitem__** are overridden.

The main part of this application is the **MatchingTracker** class. This class creates and stores tracks, and also provides matching between tracks and objects detected in the frame. The **MatchingTracker** class contains the following methods:

- **__init__** is a constructor, it creates inside itself a list of **Tracklets**.
- **add_new_track** is a method to add a new track.
- **filter_detections** is a method that filters the detection output and creates objects of type **DetectedObject** from the detection output.
- **_shape_affiinity** is a method that calculates the similarity coefficient of two objects based on their size.
- **_place_affinity** is a method that calculates the similarity coefficient of two objects based on shapes affinity.
- **_affinity** is a method that calculates the total similarity coefficient based on shape and location affinities.
- **_compute_dissimilarity_matrix** is a method for constructing a two-dimensional similarity matrix, which consists of similarity coefficient between tracks and detected objects.
- **_solve_assignment_problem** is a method to search for the best matches of tracks and detections by solving the assignment problem.
- **process_new_frame** is a public method that processes the new frame.
- **_draw_active_track** is a method for drawing existing tracks on the frame.

```
class MatchingTracker:
    """
    Class that stores and processes tracks
    """

    def __init__(self, dist_weight=0.02, shape_affinity_weight=0.5,
            place_affinity_weight=0.7):
        pass
    def add_new_track(self, detection):
        pass
    def filter_detections(self, detect_mat, threshold=0.5):
        pass
    def process_new_frame(self, frame, detections, timestamp):
        pass
    def _solve_assignment_problem(self, tracks, detections):
        pass
    def _compute_dissimilarity_matrix(self, tracks, detections):
        pass
    def _affinity(self, obj1, obj2):
        pass
    def _shape_affinity(self, obj1, obj2, weight):
        pass
    def _place_affinity(self, o1, o2, weight):
        pass
    def draw_active_tracks(self, image):
        pass
```

Further, we consider the implementation of the above methods.

4.3 Developing method to calculate the similarity coefficient between tracks and detections based on the place affinity

The method of calculating the similarity coefficient between track and detection based on shape affinity takes the last object location in the track and the detection, and the coefficient "place affinity weight score". This method calculates the similarity coefficient according to the formula presented in the algorithm described in the previous section. The result of the method is a value from 0 to 1.

```
def _place_affinity(self, obj1, obj2, weight):
    obj1_x = (obj1.x_left + obj1.x_right) * 0.5
    obj1_y = (obj1.y_top + obj1.y_bottom) * 0.5
    obj2_x = (obj2.x_left + obj2.x_right) * 0.5
    obj2_y = (obj2.y_top + obj2.y_bottom) * 0.5
    obj2_width = obj2.x_right- obj2.x_left
    obj2_height = obj2.y_top - obj2.y_bottom
    x_dist = ((obj1_x - obj2_x) ** 2) / (obj2_width ** 2)
    y_dist = ((obj1_y - obj2_y) ** 2) / (obj2_height ** 2)
    return math.exp(-weight * (x_dist + y_dist))
```

4.4 Developing method to calculate the similarity coefficient between tracks and detections based on the shape affinity

The method for calculating the similarity coefficient between track and detection based on shape affinity takes the last object location in the track and the detection, and the weight for coefficient "shape affinity weight score". This method calculates the similarity coefficient according to the formula presented in the algorithm described in the previous section. The result of the method is a value from 0 to 1.

```
def _shape_affinity(self, obj1, obj2, weight):
    obj1_width = obj1.x_right-obj1.x_left
    obj2_width = obj2.x_right-obj2.x_left
    obj1_height = obj1.y_top - obj1.y_bottom
    obj2_height = obj2.y_top - obj2.y_bottom
    w_dist = abs(obj1_width - obj2_width) / (obj1_width + obj2_width)
    h_dist = abs(obj1_height - obj2_height) / (obj1_height + obj2_height)
    return math.exp(-weight * (w_dist + h_dist))
```

4.5 Developing method to calculate the total similarity coefficient between track and detection

The **`_affinity`** method calculates the total similarity coefficient between tracks and detections using the similarity coefficients obtained with the previous two methods. The result of this function is a value from 0 to 1.

```
def _affinity(self, obj1, obj2):
    shp_aff = self._shape_affinity(obj1, obj2,
        self._shape_affinity_weight)
    mot_aff = self._place_affinity(obj1, obj2,
        self._place_affinity_weight)
    return shp_aff * mot_aff
```

4.6 Developing method to construct similarity matrix of coefficients between tracks and objects

The **`_compute_dissimilarity_matrix`** method constructs a similarity matrix where cell **`[i,j]`** contains a similarity coefficient of the track **`i`** and the detection **`j`**. To solve this assignment problem, the matrix should be square, it means that the number of tracks should correspond to the number of detections. If this condition is not true, empty rows and columns with zero elements should be added to the matrix.

```
def _compute_dissimilarity_matrix(self, tracks, detections):
```

```
    size = max(len(tracks), len(detections))
    diss_mat = np.zeros(shape=(size, size), dtype=float)
    for i in range(len(tracks)):
        for j in range(len(detections)):
            diss_mat[i, j] = 1.0 - \
                self._affinity(tracks[i][-1], detections[j])
    return diss_mat
```

4.7 Developing method for solving the assignment problem

The **_solve_assignment_problem** method searches for the best matches between the tracks and detections through solving the assignment problem [6]. The input of this method is a list of tracks and a list of detections; the output is a set of matched pairs of track identifiers (row of the similarity matrix) and bounding box identifiers (column of the similarity matrix).

```
from scipy.optimize import linear_sum_assignment

def _solve_assignment_problem(self, tracks, detections):
    dissimilarity_mat = self._compute_dissimilarity_matrix(
        tracks, detections)
    row_ind, col_ind = linear_sum_assignment(dissimilarity_mat)
    return row_ind, col_ind
```

4.8 Developing method for filtering detections before assignment

The **filter_detections** method receives the result of object detection using a deep model (for example, the SSD300 model considered in the previous practice), and creates a list of **DetectedObject** objects that will be processed. Objects for which the confidence is less than a certain threshold (0.5 by default) are discarded and are not included in the list of detected objects.

```
def filter_detections(self, detect_mat, threshold=0.5):
    detect_mat = detect_mat[0, 0, :, :]
    detections = []
    for i in range(detect_mat.shape[0]):
        # Parse one string in ie_detection_output
        conf = detect_mat[i, 2]
        if conf > threshold:
            detection = DetectedObject(
                detect_mat[i, 2],
                -1,
                -1,
                -1,
                detect_mat[i, 1],
                detect_mat[i, 3],
                detect_mat[i, 4],
                detect_mat[i, 5],
                detect_mat[i, 6])
            detections.append(detection)
    return detections
```

4.9 Developing method to create a new track

This method receives a new detection as an input, creates a new track and adds it to the track list.

```
def add_new_track(self, detection):
    track = Tracklet(detection)
    self._tracks.append(track)
```

4.10 Developing method for processing a video frame

The method receives a list of bounding boxes detected on the new frame as an input, constructs a similarity matrix, and solves the assignment problem. For the matches, the similarity coefficient for which

is above the threshold, we update the corresponding tracks. If the similarity coefficient is low, then new tracks are created for such detections.

```
def process_new_frame(self, frame, detections, timestamp):
    if self._tracks and detections:
        row_indexes, col_indexes = self._solve_assignment_problem(
            self._tracks, detections)
        # For each assignment
        for i in range(len(row_indexes)):
            row_id = row_indexes[i]
            col_id = col_indexes[i]
            # If we find existing track and existing detection
            if col_id < len(self._tracks) and col_id < len(detections):
                # Add detection to track if objects are close
                dist = 1.0 - self._affinity(
                    self._tracks[row_id][-1], detections[col_id])
                if dist > self._dist_weight:
                    self._tracks[row_id].add_new_detection(
                        detections[col_id])
                elif col_id < len(detections):
                    self.add_new_track(detections[col_id])
    else:
        # Add new tracks
        for id, detection in enumerate(detections):
            self.add_new_track(detection)
    return
```

4.11 Developing method for displaying tracks

The **draw_active_tracks** method draws tracks on the frame. Each track is drawn as follows: the centers of the bounding boxes in frames **i** and **i+1** are calculated and a line is drawn between them.

```
def draw_active_tracks(self, image):
    w, h = image.shape[:2]
    for i, track in enumerate(self._tracks):
        # Draw one track from segments
        for i in range(len(track) - 1):
            cv2.line(img=image,
                pt1=(int((track[i].x_left + track[i].x_right) * h)//2,
                    int((track[i].y_bottom + track[i].y_top) * w)//2),
                pt2=(int((track[i+1].x_left + track[i+1].x_right) * h)//2,
                    int((track[i+1].y_bottom + track[i+1].y_top) * w)//2),
                color=(0, 255, 0), thickness=3)
    return image
```

4.12 Creating entity for object tracking

The constructor of the **MatchingTracker** class creates an empty list of tracks and sets parameters for calculating the similarity coefficients of objects. These parameters are set for a specific video separately.

```
def __init__(self, dist_weight=0.02, shape_affinity_weight=0.5,
             place_affinity_weight=0.7):
    self._tracks = []
    self._dist_weight = dist_weight
    self._shape_affinity_weight = shape_affinity_weight
    self._place_affinity_weight = place_affinity_weight
    return
```

4.13 Implementing sample

4.13.1 Parsing command line options

In this practice, the following command line arguments will be required:

- Path to the input video (required).
- Path to the model weights file (required).
- Path to the model configuration file (required).
- Path to the dynamic library with custom layers (needed to infer the SSD-based models using CPU) (optional).
- Path to the file containing class names (optional).

The implementation of the command line parser using the **`argparse`** package is represented below.

```
def build_argparser():
    parser = argparse.ArgumentParser()
    parser.add_argument('-m', '--model', help = 'Path to an .xml \
        file with a trained model.', required = True, type = str)
    parser.add_argument('-w', '--weights', help = 'Path to an .bin file \
        with a trained weights.', required = True, type = str)
    parser.add_argument('-i', '--input', help = 'Path to \
        input video', required = True, type = str)
    parser.add_argument('-l', '--cpu_extension', help='MKLDNN \
        (CPU)-targeted custom layers. Absolute path to a shared library \
        with the kernels implementation', type=str, default=None)
    parser.add_argument('-c', '--classes', help = 'File containing \
        classnames', type = str, default = None)
    return parser
```

4.13.2 Implementing main function

In the file **`tracking_sample.py`** create a function **`main`** that implements the following steps:

1. Parsing command line arguments.
2. Creating an object of the **`InferenceEngineDetector`** class.
3. Creating an object of the **`MatchingTracker`** class with empirically selected affinity weight parameters for the input video.
4. Loading the video.
5. Repeating the following actions for each video frame:
   5.1. Detecting objects in the frame.
   5.2. Filtering detections using the **`tracker.filter_detections`** method.
   5.3. Tracking objects using the **`tracker.process_new_frame`** method.
   5.4. Drawing tracking results on the frame and displaying on the screen.

```
def main():
    args = build_argparser().parse_args()

    ie_detector = InferenceEngineDetector(configPath=args.model,
        weightsPath=args.weights, device=args.device,
        extension=args.cpu_extension, classesPath=args.classes)

    tracker = MatchingTracker()

    cap = cv2.VideoCapture(args.input)

    timestamp = 0
    while(cap.isOpened()):
```

```
        timestamp += 1
        _, frame = cap.read()

        detections_mat = ie_detector.detect(frame)
        detections = tracker.filter_detections(detections_mat, 0.5)

        tracker.process_new_frame(frame, detections, timestamp)

        tracks_image = tracker.draw_active_tracks(frame)

        result_image = ie_detector.draw_detection(detections_mat,
            tracks_image)

        cv2.imshow('Detections', result_image)
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

    cap.release()
    cv2.destroyAllWindows()
    return
```

5 Executing developed sample

The easiest way to execute your sample is the command line represented below.

```
python tracking_sample.py -i video.mp4 -m ssd300.xml -w ssd300.bin \
    -l "C:\Program Files
(x86)\IntelSWTools\openvino\deployment_tools\inference_engine\bin\intel64\
Release\cpu_extension_avx2.dll"
```

The **-i** argument specifies the path to the input video, the **-m** argument specifies the model configuration path, the **-w** argument specifies the model weight path, the **-c** argument specifies the path to the file containing object class names, the **-l** argument specifies the path to the dynamic library with custom layers.

The result of launching the application looks like in the paragraph below. A message about the start of the application is displayed, then a window is opened in which the video frames with the detected objects and tracks are displayed (Fig. 1).

```
[ INFO ] Start matching tracking sample
```

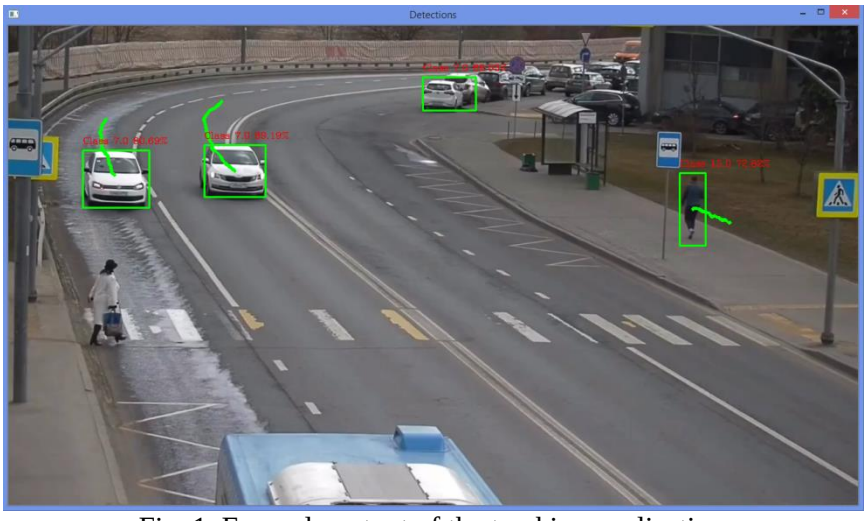

Fig. 1. Example output of the tracking application

6 Additional tasks

The developed sample contains the minimum required functionality. As additional tasks, it is proposed to provide support for the following features:

1. Displaying the total number of detected objects on each frame and the number of objects by class. We propose to display information about the most numerous three classes.
2. Limiting the number of object classes depending on the video context. For example, if a detection model trained on the PASCAL VOC dataset and we process video from crossroad camera, then only traffic objects (cars, buses, pedestrians) have to be detected and tracked.
3. Supporting for dividing tracks into “active” and “inactive”. By “inactive” we mean tracks for which the corresponding object was not found within a certain period of time (for example, within a few seconds). It is proposed to draw tracks using different colors during this time period, and after this time not to display these tracks.

It is proposed to solve these tasks independently using the documentation and examples included in the OpenVINO Toolkit.

7 Literature

7.1 Books

1. Chollet F. Deep Learning with Python. – Manning Publications Co, NY, USA, – 2017.

7.2 Further reading

2. Ramalho L. Fluent Python: Clear, Concise, and Effective Programming. – O’Reilly Media, Inc., CA, USA, 2015.
3. Kuhn H.W. The Hungarian Method for the assignment problem // Naval Research Logistics Quarterly. – 1955.
4. Kuhn H.W. Variants of the Hungarian method for assignment problems // Naval Research Logistics Quarterly. – 1956.

7.3 References

5. Open Model Zoo repository of deep models [https://github.com/opencv/open_model_zoo].
6. The assignment problem and The Hungarian algorithm [http://www.hungarianalgorithm.com].