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 №4
Solving the problem of video analytics, including detection, recognition and tracking of objects in the video

Supported by Intel

Vasiliev E.P.

Nizhny Novgorod
2020

Content

1 Introduction

In this practice, we consider the widespread problem of video analytics, namely, the problem of counting vehicles of different classes on video. We develop a solution based on the vehicle detection using deep learning models and their tracking in the video. Pre-trained deep models from the Open Model Zoo repository are used [4].

2 Guidelines

2.1 Goals and tasks

The goal of this practice is to study the problem of counting vehicles of different classes on video and develop an application to solve it using the Intel Distribution of OpenVINO Toolkit. To achieve this goal, it is required to solve the following tasks:

- Study the problem statement of counting vehicles of different classes on video and the algorithm for solving this problem.
- Develop an application based on the Inference Engine for counting vehicles. The counting result should be displayed on the original video.
- Execute and verify the developed sample.

2.2 Practice structure

First, a brief description of the problem of counting vehicles of different classes on video is provided. The algorithm for solving this task is described. Further, the application of counting vehicles based on the object detection and tracking by matching is developed step-by-step.

2.3 Recommended study sequence

The recommended practice sequence is as follows:

- Explore samples of solving different video analytics tasks using the Intel Distribution of OpenVINO Toolkit.
- Setup software environment for the Intel Distribution of OpenVINO Toolkit.
- Study the problem statement of counting vehicles of different classes on video and the algorithm for solving this problem.
- 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.

2.4 Samples of solved video analytics tasks using the Intel Distribution of OpenVINO Toolkit

Intel Distribution of OpenVINO Toolkit includes a large set of samples for solving video analytics tasks. Each sample contains detailed comments, includes a description of the implemented approach, input and output data, deep models available for this sample [3]. Below are some examples of available applications.

1. Action recognition Python demo [5]. This is the demo application for action recognition algorithm, which classifies actions that are being performed on input video.
2. Crossroad camera demo [6]. This demo provides an inference pipeline for persons' detection, recognition and reidentification. The demo uses person detection network followed by the person attributes recognition and person reidentification retail networks applied on top of the detection results.
3. Security Barrier Camera C++ Demo [7]. This demo showcases vehicle and license plate detection network followed by the vehicle attributes recognition and license plate recognition networks applied on top of the detection results.

4. Smart Classroom C++ Demo [8]. The demo shows an example of joint usage of several neural networks to detect student actions (sitting, standing, raising hand, turned around) and recognize people by faces in the classroom environment.

The number of problems to be solved using the Intel Distribution of OpenVINO Toolkit is increasing due to new demo applications.

3 Problem statement of counting vehicles on video

In this practice, the problem of counting vehicles of different classes on the video from the crossroad camera is considered. The input is a sequence of video frames. It is required to count the number of vehicles of each category ("car", "bus", "motorbike" and others) that have already been observed in the video by the current moment.

4 Algorithm for counting vehicles of different classes on video

The base algorithm for counting vehicles of different classes on video based on the object detection and tracking consists of the several steps performed processing the next received frame.

1. Object detection. To detect objects, any deep model capable of detecting vehicles of different classes ("car", "bus", "train", and "motorbike") can be used. It is required to filter out objects that do not belong to the set of vehicle classes. Filtering can be performed at the detection stage or at the stage of calculating statistics. It is also necessary to remember that the detection algorithm for the same object on different frames can predict a different class. In our solution, the class of the object is the class predicted at the last processed frame. For vehicle detection, the SSD300 model and **`InferenceEngineDetector`** class can be used.
2. Object tracking. You can use tracking-by-matching implementation of the tracker from the practice "Tracking objects on video using deep neural networks".
3. Calculating statistics for vehicles of different classes, which were observed from the beginning of the video to the current frame. To calculate statistics, it is required to create counters for each class of vehicles and to iterate over all the tracks, increasing the counters of the corresponding vehicle class.
4. Displaying frame with detected vehicles and their tracks, displaying the collected statistic information on the screen.

5 Developing the application for counting vehicles of different classes on video

5.1 File structure

For this practice, please, create two files**:** **`videoanalytics.py`** is a file containing the **`Videoanalytics`** class to calculate and display video analytics, and **`videoanalytics_sample.py`** is file containing the testing code. It is assumed that the modules **`ie_detector.py`** and **`matching_tracker.py`**, containing implementations of object detection and tracking algorithms, have already been developed during previous practices.

A class that calculates video analytics is **`Videoanalytics`**. This class provides several methods.

- **`_init_`** is a class constructor. It loads a list of vehicle classes to be counted, and the names of these classes to display on the screen.
- **`count_objects_per_classes`** is a method that counts vehicles of each class.
- **`draw_videoanalytics`** is a method for displaying the vehicle counting results on the video.

The constructor implementation is trivial, so it is proposed to develop it by yourself. The implementation of other methods will be described below.

5.2 Counting vehicles of different classes on video

Consider the implementation of the **count_objects_per_classes** method for counting the number of vehicles of each class. This method iterates over all tracks obtained as a result of tracking, and counts the number of objects of each class.

```
def count_objects_per_classes(self, tracker):
    results = {}
    for tracklet in tracker._tracks:
        # Get object class from tracklet
        classid = tracklet[-1].class_id
        # Add +1 to counter for object
        if classid in results and classid in self.classIds:
            results[classid] += 1
        else:
            results[classid] = 1
    return results
```

5.3 Displaying of vehicle counting results

The **draw_videoanalytics** method displays the results of counting vehicles. It consists of three parts.

1. Displaying detected objects of vehicle classes. It is required to iterate over all tracks.
   1.1. Searching for the last bounding box corresponding to the object location in the track.
   1.2. If the object class is in the list of vehicle classes and the object is found in the current frame, then displaying the bounding box.
2. Displaying the tracks of vehicles. It is supposed to iterate over all tracks.
   2.1. Searching for the last bounding box corresponding to the object location in the track.
   2.2. If the object class is in the list of vehicle classes and the object is found in the current frame, then iterate over all bounding box in the track, calculate their centers and draw line between neighbor frame centers.
3. Displaying statistics.
   3.1. Count the number of vehicles of each class using the **count_objects_per_classes** method.
   3.2. For each vehicle class, create a line to display the following information: “Class <class_name>: <number_of_objects>”.
   3.3. Displaying lines with counting information in the upper left corner of the frame.

```
def draw_videoanalytics(self, frame, tracker):
    (h, w) = frame.shape[:2]

    # Draw detections of chosen classes
    for tracklet in tracker._tracks:
        # Get object class from tracklet
        classid = tracklet[-1].class_id
        if classid in self.classIds:
            # Draw bbox of the last detection
            x_left = int(tracklet[-1].x_left * w)
            y_bottom = int(tracklet[-1].y_bottom * h)
            x_right = int(tracklet[-1].x_right * w)
            y_top = int(tracklet[-1].y_top * h)
            cv2.rectangle(frame, (x_left, y_bottom), (x_right, y_top),
                (0, 255, 0), 2)
    # Draw tracklets of chosen classes
    for track in tracker._tracks:
        # Get object class from tracklet
        classid = track[-1].class_id
```

```
        if classid in self.classIds:
        # Draw one tracklet from segments
            for i in range(len(track)-1):
                x1 = int((track[i].x_left+track[i].x_right)*w)// 2
                y1 = int((track[i].y_bottom+track[i].y_top)*h)// 2
                x2 = int((track[i+1].x_left+track[i+1].x_right)*w)// 2
                y2 = int((track[i+1].y_bottom+track[i+1].y_top)*h)// 2
                cv2.line(img=frame, pt1=(x1,y1), pt2=(x2,y2),
                    color=(0, 255, 0), thickness=3)
    # Draw statistics
    counts = self.count_objects_per_classes(tracker)
    for i, elem in enumerate(counts):
        if self.classes:
            id = int(elem)
            text = 'Class {}: {} objects'.format(
                self.classes[id], counts[elem])
        else:
            text = 'Class {}: {} objects'.format(elem, counts[elem])
        text_pos = (0, i * 30 + 30)
        cv2.putText(frame, text, text_pos,
            cv2.FONT_HERSHEY_COMPLEX, 1.0, (0, 255, 255), 2)
    return frame
```

5.4 Implementing sample

5.4.1 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 (it is required to infer the SSD-based models on CPU) (optional).
- Path to the file containing list of detected object classes (required).
- Identifiers of vehicle classes (required). This is a list of several numbers separated by a space (for example “2 6 7 14 19”).

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('-d', '--device', help='Specify the target \
        device to infer on; CPU, GPU, FPGA or MYRIAD is acceptable. \
        Sample will look for a suitable plugin for device specified \
        (CPU by default)', default='CPU', type=str)
    parser.add_argument('-c', '--classes', help='File containing classes \
        names', type=str, default=None)
    parser.add_argument('-c_d', '--classes_detected', help='List of \
```

```
        classes to do videoanalysis', type=int, nargs='+',
        default='2 6 7 14 19')
    return parser
```

5.4.2 Implementing main function

The **main** function is very similar to the **main** function from the object tracking practice, with a small difference in that the **draw_videoanalytics** method is used here to display the results of video analytics.

In the file **video_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. Creating an object of the **Videoanalytics** class with the necessary parameters.
5. Loading the video.
6. Performing actions for each video frame:
   6.1. Detecting objects in the image.
   6.2. Filtering detections using the **tracker.filter_detections** method.
   6.3. Tracking objects using the **tracker.process_new_frame** method.
   6.4. Displaying detections and tracks of vehicles on the frame and displaying the statistics on the frame using the **videoanalytics.draw_videoanalytics** method.

```
def main():
    log.basicConfig(format="[ %(levelname)s ] %(message)s",
                    level=log.INFO, stream=sys.stdout)
    args = build_argparser().parse_args()

    log.info("Start videoanalytics sample")

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

    tracker = MatchingTracker()

    videoanalytics = Videoanalytics(args.classes_detected, args.classes)

    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)

        result_image = videoanalytics.draw_videoanalytics(frame, tracker)

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

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

6 Executing developed sample

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

```
python videoanalytics_sample.py -i video.mp4 -m ssd300.xml -w ssd300.bin \
  -c pascal_voc.txt -c_d 2 6 7 14 19
```

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 **-c_d** argument specifies a list of vehicle class identifiers, the **-l** argument specifies the path to the dynamic library with custom layers.

The result of the application execution is as follows. A message about the start of the application is displayed, then a window is opened in which the video frames with the detected objects, tracks and counting statistics are displayed (Fig. 1).

```
[ INFO ] Start videoanalytics sample
```

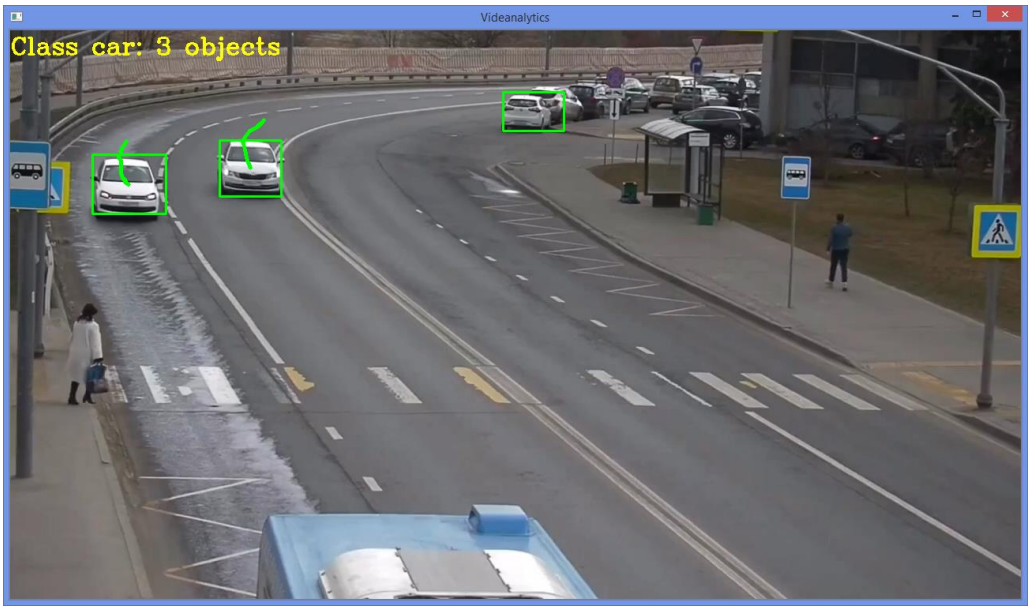

Fig. 1. Example output of the calculating statistics for class "car"

7 Additional tasks

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

1. The scheme with storing information about all detected objects in tracks is not effective. It is proposed to filter non-vehicles immediately after detection and track vehicles.
2. The detector on different frames for the same object can predict various object classes. It is supposed to implement a scheme for determining the object class based on the full track information, not just the last frame. The easiest scheme is the voting one, when the object class is determined by the majority of the bounding boxes in the track.

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

8 Literature

8.1 Books

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

8.2 Further reading

2. Ramalho L. Fluent Python: Clear, Concise, and Effective Programming. – O'Reilly Media, Inc., CA, USA, 2015.

8.3 References

3. Inference Engine demos [https://docs.openvinotoolkit.org/latest/omz_demos_README.html].
4. Open Model Zoo repository of deep models [https://github.com/openvinotoolkit/open_model_zoo].
5. Action recognition Python demo [https://docs.openvinotoolkit.org/latest/omz_demos_python_demos_action_recognition_README.html].
6. Crossroad camera demo [https://docs.openvinotoolkit.org/latest/omz_demos_crossroad_camera_demo_README.html].
7. Security barrier camera demo [https://docs.openvinotoolkit.org/latest/omz_demos_security_barrier_camera_demo_README.html]
8. Smart Classroom C++ Demo [https://docs.openvinotoolkit.org/latest/omz_demos_smart_classroom_demo_README.html].