Universidad Politécnica de Yucatán

Data Engineering
Data Mining

Professor:
Victor Ortiz Santiago

Student:
Christopher Aaron Cumi Llanes

Documentation for Tracking Algorithm

Due Date:
06/12/2023

```
[1]  1 !nvidia-smi

Thu Dec  7 00:15:00 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 525.105.17   Driver Version: 525.105.17   CUDA Version: 12.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   44C    P8     9W /  70W |      0MiB / 15360MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

The command !nvidia-smi is a utility from NVIDIA, used primarily on systems with NVIDIA GPUs (Graphics Processing Units). It stands for "NVIDIA System Management Interface."

```
[2]  1 import os
     2 HOME = os.getcwd()
     3 print(HOME)

/content
```

This line imports the os module, which provides a portable way to use operating system-dependent functionality.

HOME = os.getcwd(): This line uses the getcwd() function from the os module. getcwd stands for "get current working directory." This function returns the current working directory of the Python script. The result is assigned to the variable HOME.
print(HOME): This line prints the value of the HOME variable. This will output the full path of the current working directory where the script is being executed.

```
[3]  1 !pip install -q gdown
     2 %cd {HOME}
     3 !gdown '1J8nFASaJS4Ca_RkX9LztvC4V4knEN6ZX'

/content
Downloading...
From: https://drive.google.com/uc?id=1J8nFASaJS4Ca_RkX9LztvC4V4knEN6ZX
To: /content/test_personas.mp4
100% 2.07M/2.07M [00:00<00:00, 163MB/s]
```

This line installs the gdown package using pip, Python's package installer. The gdown tool is used for downloading files from Google Drive.

The second line changes the current working directory to the one stored in the variable HOME.
%cd is a magic command used in Jupyter notebooks and similar environments to change the current working directory.

{HOME} is a variable that contains the path to the desired directory, which was previously set to the current working directory of the Python script.

The third line uses gdown to download a file from Google Drive.
The string '1J8nFASaJS4Ca_RkX9LztvC4V4knEN6ZX' is the unique identifier (or 'file ID') for a file on Google Drive. This ID is part of the shareable link of a file stored on Google Drive.

```
[4]  1 SOURCE_VIDEO_PATH = f'{HOME}/test_personas.mp4'
```

This line of code is creating a string that represents a file path to a video file named test_personas.mp4. The file path is constructed using an f-string, which is a way to embed expressions inside string literals, using curly braces {}.

SOURCE_VIDEO_PATH: This variable is being assigned the file path that is constructed in this line of code.

f'{HOME}/test_personas.mp4': This is an f-string (denoted by the prefix f before the opening quote). F-strings are used in Python for string formatting.

- {HOME}: This is a placeholder within the f-string. The value of the HOME variable will be inserted here.
- /test_personas.mp4: This is a string literal specifying the relative path (from HOME) to the video file.

```
[5]  1 !pip install ultralytics
     2
     3 from IPython import display
     4 display.clear_output()
     5
     6 import ultralytics
     7 ultralytics.checks()

Ultralytics YOLOv8.0.223 🚀 Python-3.10.12 torch-2.1.0+cu118 CUDA:0 (Tesla T4, 15102MiB)
Setup complete ✅ (2 CPUs, 12.7 GB RAM, 26.9/78.2 GB disk)
```

Install the ultralytics package using pip. This package likely contains tools or functionalities related to Ultralytics' machine learning and computer vision solutions.

Import the display module from IPython and clear the output of the current cell. This is useful in interactive environments like Jupyter notebooks for maintaining a clean output area.

Import the ultralytics module. This module provides access to functionalities offered by the Ultralytics package.

Perform initial checks or setups required by the ultralytics tools. The exact nature of these checks is dependent on the specifics of the ultralytics package.

```
[6]  1 !pip install supervision
     2
     3 from IPython import display
     4 display.clear_output()
     5
     6 import supervision as sv
     7 print('supervision.__version__:', sv.__version__)

     supervision.__version__: 0.17.0
```

Install the 'supervision' package using pip.

Import the display module from the IPython package, which is used in interactive computing environments like Jupyter notebooks. Then clear the output of the current cell. This action is useful for keeping the output area clean and more readable, especially after installation commands.

Import the 'supervision' package and alias it as 'sv'. This allows us to use the functionalities provided by this package more concisely.

Print the version of the 'supervision' package. This is helpful for debugging, ensuring compatibility, as different versions of the package might have different functionalities or requirements.

```
[7]  1 MODEL = 'yolov8x.pt'

[8]  1 from ultralytics import YOLO
     2
     3 model = YOLO(MODEL)
     4 model.fuse()

     Downloading https://github.com/ultralytics/assets/releases/download/v0.0.0/yolov8x.pt to 'yolov8x.pt'...
     100%|██████████| 131M/131M [00:00<00:00, 270MB/s]
     YOLOv8x summary (fused): 268 layers, 68200608 parameters, 0 gradients, 257.8 GFLOPs
```

Define a variable 'MODEL' and assign it the string 'yolov8x.pt'. This string represents the filename of the model, that is a pre-trained model used for object detection.

Import the YOLO class from the ultralytics package.

Instantiate a YOLO model object using the pre-defined 'MODEL' variable, which contains the filename of the model ('yolov8x.pt'). This line creates a model object from the specified model file, readying it for performing object detection tasks.

Call the 'fuse' method on the model object. This method optimizes the model by fusing certain layers and operations, potentially improving performance by reducing the computational load during inference.

```
[9]  1 # dict maping class_id to class_name
     2 CLASS_NAMES_DICT = model.model.names
     3
     4 # Class_ids of interest
     5 selected_classes = [0]

[10] 1 import supervision as sv
     2 import numpy as np
```

Retrieve the class names from the 'model' object and store them in 'CLASS_NAMES_DICT'. In the context of object detection models like YOLO, 'model.model.names' refers to a list of class names that the model is trained to recognize, such as 'person', 'car', etc. The variable 'CLASS_NAMES_DICT' suggests storing these class names, although it is actually a list, not a dictionary.

Define a list 'selected_classes' and initialize it with the value [0]. This list is intended to specify indices of classes that are of interest or relevant for further processing. In this case, '0' suggests that only the first class in 'CLASS_NAMES_DICT' is selected, in this case, humans.

```
[11]  1 # create frame generator
      2 generator = sv.get_video_frames_generator(SOURCE_VIDEO_PATH)
      3 # create instance of BoxAnnotator
      4 box_annotator = sv.BoxAnnotator(thickness=1, text_thickness=1, text_scale=.5)
      5 # acquire first video frame
      6 iterator = iter(generator)
      7 frame = next(iterator)
      8 # model prediction on single frame and conversion to supervision Detections
      9 results = model(frame, verbose=False)[0]
     10
     11 # convert to Detections
     12 detections = sv.Detections.from_ultralytics(results)
     13 # only consider class id from selected_classes define above
     14 detections = detections[np.isin(detections.class_id, selected_classes)]
     15
     16 # format custom labels
     17 labels = [
     18     f"{CLASS_NAMES_DICT[class_id]} {confidence:0.2f}"
     19     for _, _, confidence, class_id, _ in detections
     20 ]
     21
     22 # annotate and display frame
     23 anotated_frame=box_annotator.annotate(scene=frame, detections=detections, labels=labels)
     24
     25 %matplotlib inline
     26 sv.plot_image(anotated_frame, (16,16))
```

Create a generator that yields frames from the video specified in SOURCE_VIDEO_PATH. This is done using the 'get_video_frames_generator' function from the 'supervision' package.

Create an instance of BoxAnnotator from the 'supervision' package. This object will be used to draw bounding boxes on the video frames. The parameters define the thickness of the boxes and text, as well as the scale of the text.

Initialize an iterator for the frame generator and acquire the first video frame. This step is preparing to process the video frame-by-frame.

Use the YOLO model to perform object detection on the single frame. The 'model' object is called with the frame as input. The 'verbose=False' argument suppresses detailed logging. The results are then indexed to get the detection results.

Convert the results from the model to 'Detections' format defined in the 'supervision' package. This standardizes the results for further processing.

Filter the detections to include only those with class IDs specified in 'selected_classes'. This step allows for focusing on specific types of detected objects.

Format custom labels for each detection. This involves creating a string for each detection that includes the class name and the confidence level.

Annotate the frame with the detected bounding boxes and labels using the 'annotate' method of the BoxAnnotator instance.

We setup the environment for displaying images inline in a Jupyter notebook.

And then we display the annotated frame using 'plot_image' function from the 'supervision' package, with a specified figure size.

```
[12]  1 # settings
      2 LINE_START = sv.Point(50, 1500)
      3 LINE_END = sv.Point(3840-50, 1500)
      4
      5 TARGET_VIDEO_PATH = f'{HOME}/test_personas-result-with-counter.mp4'
```

```
[13]  1 sv.VideoInfo.from_video_path(SOURCE_VIDEO_PATH)

      VideoInfo(width=1280, height=720, fps=25, total_frames=341)
```

Define settings for a line that will be used in the video processing. This line is used for counting objects. 'LINE_START' and 'LINE_END' define the starting and ending points of the line, respectively.

'LINE_START' is set to a Point object with coordinates (50, 1500). This represents the starting point of the line on the video frame. 'sv.Point' is a function or class from the 'supervision' package that creates a point object.

'LINE_END' is set to a Point object with coordinates (3840-50, 1500). This represents the end point of the line. The calculation '3840-50' extends the line almost across the width of the frame, assuming a frame width of 3840 pixels.

Define a variable 'TARGET_VIDEO_PATH' that contains the path where the processed video will be saved. This path is constructed using an f-string with the 'HOME' variable (which holds the current working directory) and appending the filename 'test_personas-result-with-counter.mp4'. This modifies the output version of the 'test_personas.mp4' video, with added object detection and counting annotations.

```
[14]  1 # create BYTETracker instance
      2 byte_tracker = sv.ByteTrack(track_thresh=0.25, track_buffer=30, match_thresh=0.8, frame_rate=30)
      3
      4 # create VideoInfo instance
      5 video_info = sv.VideoInfo.from_video_path(SOURCE_VIDEO_PATH)
      6
      7 # create frame generator
      8 generator = sv.get_video_frames_generator(SOURCE_VIDEO_PATH)
      9
     10 # create LineZone instance, it is previously called LineCounter class
     11 line_zone = sv.LineZone(start=LINE_START, end=LINE_END)
     12
     13 # create instance of BoxAnnotator
     14 box_annotator = sv.BoxAnnotator(thickness=1, text_thickness=1, text_scale=.5)
     15
     16 # create instance of TraceAnnotator
     17 trace_annotator = sv.TraceAnnotator(thickness=4, trace_length=50)
     18
     19 # create LineZoneAnnotator instance, it is previously called LineCounterAnnotator class
     20 line_zone_annotator = sv.LineZoneAnnotator(thickness=4, text_thickness=4, text_scale=2)
```

Create an instance of BYTETracker from the 'supervision' package. BYTETracker is used for real-time multi-object tracking. Various parameters are set to configure the tracker:
- 'track_thresh': Threshold for the tracker (0.25 in this case).
- 'track_buffer': Buffer size for tracking (30 frames).
- 'match_thresh': Threshold for matching objects (0.8).
- 'frame_rate': The frame rate of the video (30 frames per second).

Create a VideoInfo instance using the source video path. VideoInfo stores metadata about the video, such as frame dimensions, duration, and other relevant information.

Re-initialize the frame generator to yield frames from the source video. This generator will be used for processing each frame of the video.

Create a LineZone instance, previously known as LineCounter class. This object represents a line zone in the video frame, which is used for counting objects. The line is defined by its start and end points, set earlier as 'LINE_START' and 'LINE_END'.

Create an instance of BoxAnnotator for drawing bounding boxes on objects in the video frames. The thickness of the boxes and text, as well as the scale of the text, are specified.

Create an instance of TraceAnnotator. This is used for drawing traces or paths that objects follow in the video. The thickness of the trace and its length (number of frames) are specified.

Create an instance of LineZoneAnnotator, previously known as LineCounterAnnotator class. This annotator is specifically used for annotating the LineZone in the video, such as for highlighting the line and displaying counts. The thickness of the line, text, and the scale of the text are specified.

```
[14]
# define call back function to be used in video processing
def callback(frame: np.ndarray, index:int) -> np.ndarray:
    # model prediction on single frame and conversion to supervision Detections
    results = model(frame, verbose=False)[0]
    detections = sv.Detections.from_ultralytics(results)
    # only consider class id from selected_classes define above
    detections = detections[np.isin(detections.class_id, selected_classes)]
    # tracking detections
    detections = byte_tracker.update_with_detections(detections)
    labels = [
        f"#{tracker_id} {model.model.names[class_id]} {confidence:0.2f}"
        for _, _, confidence, class_id, tracker_id
        in detections
    ]
    annotated_frame = trace_annotator.annotate(
        scene=frame.copy(),
        detections=detections
    )
    annotated_frame=box_annotator.annotate(
        scene=annotated_frame,
        detections=detections,
        labels=labels)

    # update line counter
    line_zone.trigger(detections)
    # return frame with box and line annotated result
    return  line_zone_annotator.annotate(annotated_frame, line_counter=line_zone)
```

Define a callback function 'callback' to be used during video processing. This function takes a frame and its index as inputs and returns the annotated frame as output.

Perform object detection on the given frame using the YOLO model. The detection results are obtained and converted into a format suitable for the supervision library ('sv.Detections').
Filter the detections to include only those with class IDs specified in 'selected_classes', which was defined earlier. This focuses the tracking on specific object types.
Update the BYTETracker with the current detections. The tracker maintains the identities of objects across frames, providing a unique ID for each tracked object.

Generate labels for each detection. Each label includes a unique tracker ID, the class name of the detected object, and the detection confidence level.
Annotate the frame with traces using the 'trace_annotator'. This annotator draws lines indicating the paths that tracked objects have followed.
Further annotate the frame with bounding boxes and labels using the 'box_annotator'. This annotator adds visual bounding boxes and labels to the frame to highlight detected and tracked objects.
Return the annotated frame.

```
# process the whole video
sv.process_video(
    source_path = SOURCE_VIDEO_PATH,
    target_path = TARGET_VIDEO_PATH,
    callback=callback
)
```

Process the entire video using the 'process_video' function from the 'supervision' package. This function applies the defined 'callback' function to each frame of the video.

'source_path': Specifies the path to the source video file. This is the video that will be processed, as defined earlier in 'SOURCE_VIDEO_PATH'.

'target_path': Specifies the path where the processed video will be saved. This is the output video file, as defined earlier in 'TARGET_VIDEO_PATH'. The processed video will include all the annotations and tracking information added by the 'callback' function.

'callback': The function to be applied to each frame of the video. This is the 'callback' function defined earlier, which includes model predictions, tracking updates, and annotations for each frame.