Lab Guide
YOLO Object Segmentation

Content Description

The following document describes a YOLO object segmentation implementation in python environment.

Lab Description

In this example, we will capture RGB images and aligned Depth images from RealSense camera and use these to segment objects of interest, as well as computing the distances to these objects. The process is shown in Figure 1.

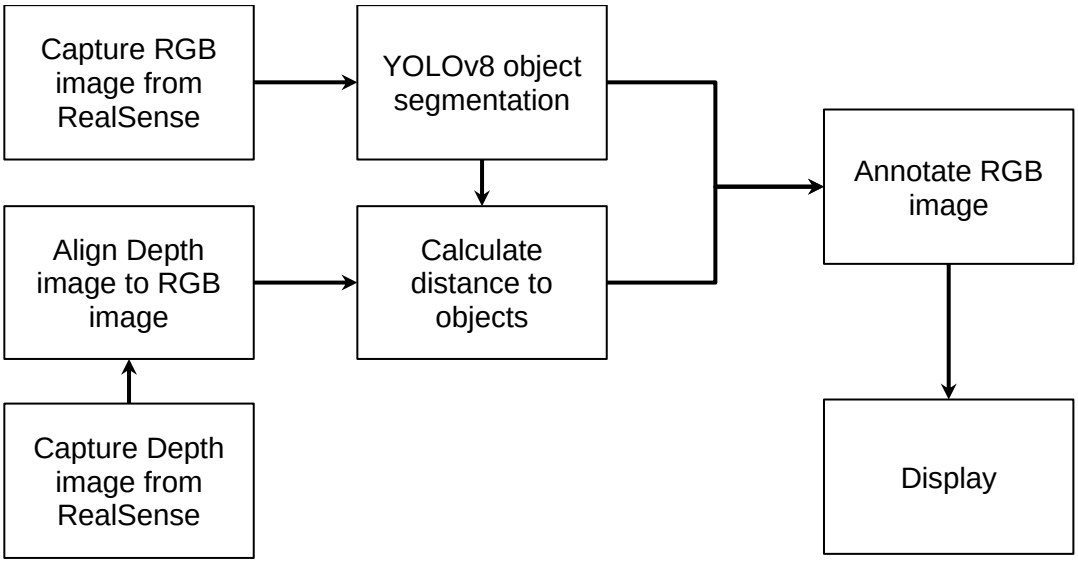

Figure 1. Component Diagram

Python

Running the example

1. Check User Manual – Software Python for details on deploying python scripts to QCar2. The output of the **cv2.imshow()** function should look like Figure 2.

2. Note that when running the script for the **first time**, the QCar2 needs to be **connected to internet**, as the trained PyTorch model will be downloaded from host. Then the downloaded model will be converted to a TensorRT engine to improve inference time, which can take up to 20 minutes.

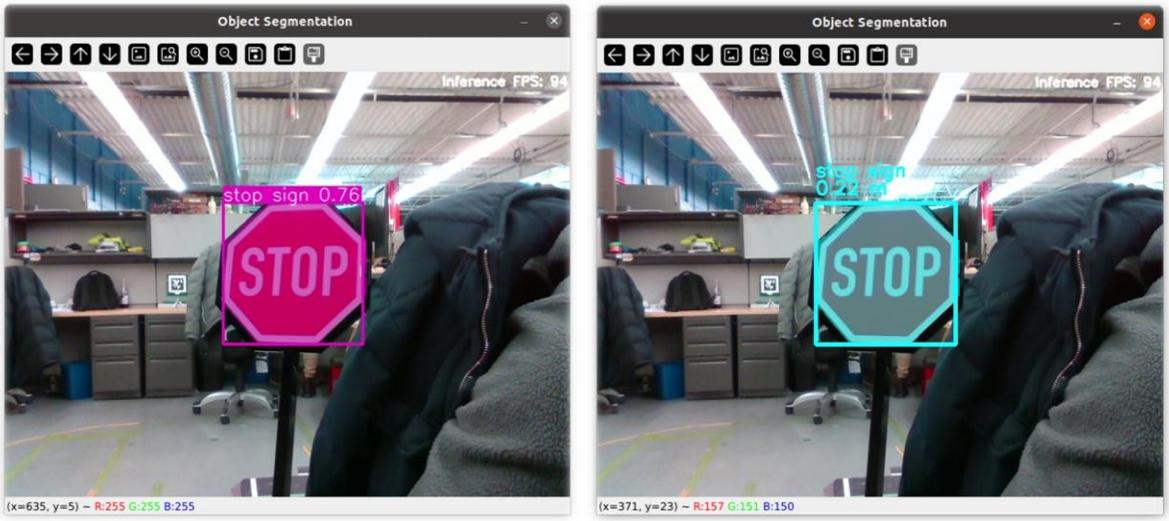

Figure 2. Annotated Camera Feed Before Post-processing (left) and After (right).

Details

1. Initialization

```
myYolo = YOLOv8(modelPath = 'path/to/model',
                imageHeight= imageHeight,
                imageWidth = imageWidth)
```

To initialize the YOLOv8 model, an optional path to a pretrained model can be provided. Otherwise, the default yolov8s-seg model trained by Quanser will be loaded, which can reliably detect other QCars. In addition, height and width of the input video stream should be provided, as the model will be optimized for that specific input size. After initialization, an **ultralytics.YOLO()** object is created and stored under the **myYolo.net** attribute.

When the script is being executed for the first time, the default model will be downloaded and then converted to a TensorRT engine for optimized inference time. The entire process can take up to 20 minutes.

2. Pre-processing and Prediction

```
rgbProcessed = myYolo.pre_process(QCarImg.rgb)
predecion = myYolo.predict(inputImg = rgbProcessed,
                           classes = [2,9,11],
                           confidence = 0.3,
                           half = True,
                           verbose = False
                           )
```

Before prediction takes place, the input image needs to be pre-processed into dimensions specified during initialization (if it is not so already).

The processed RGB image should be used as input for the **predict()** method. The argument **classes** specify the class IDs of the objects that should be included in the prediction. The class names associated with the class IDs can be accessed in the **name** attribute of **myYolo.net** as follows:

```
>>> myYolo.net.names
{0: 'person', 1: 'bicycle', 2: 'car', ...}
```

In addition, **confidence** defines the minimum confidence the detected objects, and **half** controls whether the model should predict using half floating-point precision, which can improve inference speed with minimal impact on accuracy.

The output of the **predict()** method is an **ultralytics Results** object, which contains the detected classes, as well as the associated pixels and bounding box locations. For more information, please refer to ultralytics documentation.

3. Post-processing and render

```
processedResults=myYolo.post_processing(alignedDepth = QCarImg.depth,
                                        clippingDistance = 5)
# annotatedImg=myYolo.render()
annotatedImg=myYolo.post_process_render(showFPS = True)
```

The **post_processing()** method takes an optional depth image as input to calculate distance of detected objects. This is done by applying the segmentation mask of each detected on the aligned depth image and computing the average depth value in these regions. The optional **clippingDistance** argument can be used to define the distance beyond which the depth value would be discarded. When no depth images are provided, the distance calculation will be skipped.

In addition, the **post_processing()** method also computes the traffic light status in the **check_traffic_light()** method, should a traffic light be detected. This is done by isolating the regions of the three lights relative to the bounding box of the traffic light. Then, the brightness of the colors in these regions can be compared and the

traffic light status can be determined. In this way, the method is more robust to varying lighting conditions.

The **render()** method returns the annotated RGB input that includes the object class, confidence, bounding box, and segmentation mask. This method utilizes the **plot()** method of the **ultralytics Results** object, for more options to customize the output, please refer to ultralytics documentation. The **render()** method can be used right after prediction**,** but the **post_process_render()** method can only be used after **post_processing()** is called. Annotated images from **post_process_render()** also includes distances to detected objects and traffic light status, as shown in Figure 1.

4. Performance considerations

The **render()** and **post_process_render()** methods are time-consuming processes, as their operations are not optimized. For practical self-driving applications, please consider not rendering the predictions and post-processing results for reduced computation time.