Guide for Yolop (code and installation)

It is recommended that a virtual environment is used for installing libraries and running the code.

Source Repository

The code is taken from this repository: Repo
This repository is owned by the authors of the original YOLOP Paper which can be accessed here: Paper

Pre-existing environment:

Python Version: 3.9.5
Operating System: Windows 11
GPU: NVIDIA GeForce RTX 3080 Ti
CUDA version: 11.5

For a different environment please make sure the inter-compatibility of the GPU, CUDA version, and the Python Version. Moreover, make sure that you install the correct version of `torch` which is built with the closest version of your CUDA version so that `torch` is able to detect the GPU. In our case, we use `torch` version `torch==1.11.0+cu113` which is built with CUDA version 11.3 as no version of `torch` was built with CUDA version 11.5.

The code is based on PyTorch and all dependencies are mentioned in the `requirements.txt` file. For a different environment, edit the `requirements.txt` file appropriately.

Setting the virtual environment and installing dependencies.

To set up an empty virtual environment, follow the steps mentioned here: VENV
Be sure to activate the virtual environment.

1. Before installing the dependencies through `requirements.txt` please remove the lines that mention `PyOpenGL` and `pyzed`. We use the ZED Mini Stereo Camera for capturing real time images.
2. To use ZED camera install the ZED SDK and follow the instruction mentioned here: pyzed. If you face the `ImportError` then follow the solution mentioned here: Solution
3. Install all the remaining dependencies by running:

   ```
   pip install -r requirements.txt
   ```

Code

`main.py`

This file contains the code for taking the image in real time from the attached ZED Camera and producing the results.

`utility.py`

This file contains the wrapper class `Yolop` and all the helper functions that are needed to produce the results.

The `Yolop` class

The `Yolop` class wraps around the model already present in the original repository and serves as a data pipeline that takes in the image one at time and produces the results. The code is optimized only for taking in real time images and expects no other form of input such as a video or a directory that contains images.

The instance variables are:

- `device` : This variable containes the device string where the model is instanciated. It can be either `'cpu'` or `'cuda'` depending on whether the GPU is detected by PyTorch. It is automatically determined.
- `model` : This variable contains the pretrained YOLOP model.
- `transform` : a `torchvision.transforms.Compose` object that will be used to convert an image into `torch.Tensor` object with normalized values.

`detect` method

The main function that is used for detecting for performing tasks in the `detect` method.

Arguments:

- `image` : an RGB image in the form of a `numpy.ndarray` having shape `(H, W, C)`.
- `visualize` : boolean variable that controls whether the resultant image should have the annotations of the detected objects and lanes. If set to `True` then the result will have the annotations and is likely to drop the computation speed in terms of Frames Per Second (FPS). Default value is `True`.
- `colors` : A list that contains the color values of the different classes that have to be annotated. The \(i^{th}\) element of this list should be the color corresponding to the \(i^{th}\) class. The class list can be obtained by using the `names` attribute of the `model` instance variable of the object. The default value is `None`.
- `return_latency` : A boolean variable that controls if latency statistics have to be returned for producing the results. If set to `True` the result will include a dictionary that has the latency statistics.

Returns:
The method returns a list consisting of five elements. Let those elements be : `[boxes, area_mask, lane_mask, latency, annotated_image]`. The details of the elements are as follows:

- `boxes` : If no objects are detected in the frame/image, then this will be set to `None`. If objects are detected then this will be a `numpy.ndarray` with shape `(n_objs, 6)` where `n_objs` are the number of objects detected in the image. Hence each object will have 6 corresponding values associated to it. The values are `[top_left_x, top_left_y, bottom_right_x, bottom_right_y, confidence, class_index]` where the first four values are the pixel coordinates of the rectangle that defines the object box. `confidence` is the probability of the object being there. `class_index` is the index of the predicted class, it will be 0 in most cases as the model is only able to detect objects and not classify them. To classify objects, furthur training will be required, the details of which can be read from here: MultiClass
- `area_mask` : This will be a binary image of the same shape as the input `image` which specifies which pixels are classfied under drivable area. A pixel value of 1 indicates that it

falls in the drivable area.
- `lane_mask` : This will be a binary image of the same shape as the input `image` which specifies which pixels are classfied as lanes. A pixel value of 1 indicates that it is part of a lane.
- `latency` : If `return_latency` is `False`, then this will be set to `None`, else, this will be a dictionary that has the latency statistics. The keys of which are the following:
    - `model_pass` : the corresponding value has the time taken in seconds to perform a forward pass of the image.
    - `obj_det` : the corresponding value has the time taken in seconds to perform object detection on the raw result obtained by forward pass. The operations include Non-Maximum Supression and Intersection-Over-Union computations.
    - `area_seg` : the corresponding value has the time taken in seconds to compute the drivable area segmentation mask.
    - `lane_det`: the corresponding value has the time taken in seconds to compute the lane segmentation mask.
    - `total`: the corresponding value has the time taken in seconds to perform all the computations from start to end. This does not include the time to annotate the results in the image.
- `annotated_image` : If `visualize` is set to `False`, then this will be the same image as the input `image`, else, this will be the image having the objects, driveable area, and lanes annotated.

Helper Functions

non_max_suppression method

This function returns the supressed result on the raw output obtained from the model. Main arguments are:

- `conf_thres` : The minimum confidence value for a detected object to be considered in the final result. The lower the value, more objects will be detected and spurious results will be there. The higher the value, very less objects will be detected. Default value is 0.25.
- `iou_thres` : The maximum value/proportion for a detected object to be intersecting with other objects. All objects having higher iou value than `iou_thres` will not be considered and filterd out. If this value is set very high (closer to 1) then many spurious results may be obtained, if set to too low, then only the most distinct object will be considered. Default value is 0.45.

You can fine tune these arguments to acheive desired results.

Guide for YoloV5

Source Repository

The code is taken from this repository: Repo

Setup

You can either clone the repository or download the zip file and extract the code into a local directory. After extracting the files, download the weights of the model from here: Weights

Place the downloaded weights in the `weights` directory of the extracted files. For more information refer to the source repository.

If you face an error that says:

```
AttributeError: 'Upsample' object has no attribute 'recompute_scale_factor
```

Then follow this solution here: Error Fix

Code

The usage is simple and the example can be referred from here: Example

`main.py`

This file contains the code for taking the image in real time from the attached ZED Camera and producing the results.

`OBJ_DETECTION` class

The main class is `OBJ_DETECTION`. The constructor takes two arguments, the first being the path to the weights and the second being the list of classes to predict. The example provided is comprehensive and describes the usage.

To obtain results, call the `detect` method of the object, it accepts a BGR image as a `numpy.ndarray` with shape `(H, W, C)`. It returns a list of dictionaries with each element dictionary corresponding to an object detected in the image. The keys of the dictionary are as follows:

- `label`: the corresponding value is the class label of the predicted object
- `score` : the corresponding value is the probability of the predicted object being there
- `bbox` : A list of tuples `[(bottom_left_x, bottom_left_y),(top_right_x, top_right_y)]` which specify the pixel coordinates of the rectangle that contains the object.