Exercise 1, part 2: Semantic Segmentation

1 Introduction

In this exercise you are asked to implement semantic segmentation for the FreiCar. In particular the goal of this exercise is to train a CNN that segments the classes cars, road and junctions. Further, we attach another head to the network that is capable to regress lane centerlines.

A code repository is provided that you have to complete in order to solve the task. We will not ask you to implement a network architecture from scratch.

A brief overview of the codebase is contained in the repository README.md

2 Tasks

You will train and evaluate the Fast-SCNN [1] semantic segmentation architecture, which is suited for efficient computation on robotic platforms. To solve this task, we prepared a repository here (which is already cloned into your freicar_ws/src folder):

https://aisgit.informatik.uni-freiburg.de/vertensj/freicar_base/-/tree/master/freicar_exercises/01-02-semantic-segmentation-exercise

![RGB Image](image1.jpg) ![GT Semantic Segmentation](image2.jpg)

Figure 1: Annotated Semantic Segmentation Example
2.1 Train Semantic Segmentation Network (4P)

You have to implement 2 code segments in order to successfully train your semantic segmentation model:

- Dataloader in `dataloader/freicar_dataloader.py`: We ask you to implement the dataloader `__getitem__()` function to support the indexing such that `dataset[i]` can be used to get i-th sample. Also, implement the `__len__()` function so that `len(dataset)` returns the size of the dataset.

- Train loop in `train.py`: In the train loop, you have to access the training samples (image and annotations) that you obtain from the dataloader, zero all previous gradients, do the model forward-pass, calculate the loss (classification loss and regression loss), backpropagate the gradients and perform one optimizer step.

An example of a GT Semantic Segmentation and the corresponding RGB image is depicted on Fig. 1.

Overall the segmentation GT data involve 4 classes:

- 0: Background
- 1: Road
- 2: Junction
- 3: Car

2.2 Semantic Segmentation Evaluation (4P)

Write a function that calculates the IoU score using the evaluation and training datasets every N epochs during training. Plot both IoU scores, belonging to the training and evaluation split over time. Also show qualitative results of your segmentation performance in the report.

2.3 Add Lane Regression Branch (4P)

So far we trained a pure semantic segmentation network. However for autonomous driving it is important to differentiate between multiple lanes. The dataset contains additional lane regression ground truth annotations. Here, a pixel has a value of 255 if it lies directly on a centerline of a lane. The value decreases with an exponential decay the further it is located from the centerline.

Go into the network definition and add a additional `Classifier` to the network that has just one output layer without any activation unit at the end. Both the semantic segmentation `Classifier` and the new `Classifier` should share their input feature maps.

Now train the new network branch (`Classifier`) to predict the lane regression values alongside with the normal semantic segmentation. You should use a standard L1 Loss for this regression task. Note that you may want to assign different weights to both losses in order to compensate for their magnitudes.

An example of the lane regression GT and the corresponding RGB image is depicted on Fig. 2.

2.4 Semantic Segmentation & Lane Regression Evaluation (4P)

Similar to the previous evaluation task you are asked to plot the network performance using the evaluation and training split each N epochs. However, this time also report the L1-error of the lane center regression task.

Furthermore you should present qualitative results showing the predicted lane regressions and semantic segmentations for some input images.

2.5 Write ROS node (6P)

2.5.1 Publish Segmentation & Lane Regression (2P/6P)

Write a ROS node that subscribes to the image topic and publishes the semantic segmentation and lane regression on distinct topics.
We provide the class `birdseyeTransformer` with the function `birdseye` which applies an inverse perspective mapping to the input image. An example of an output can be seen on Fig. 2.

Map your lane regression to the bird’s eye view using this function and threshold the result to get just pixels belonging to the center-lines. Every coordinate of the pixel in the bird’s eye view is in centimeters and are represented in the tf-frame: `/freicar_1/base_link` (If you are not familiar with the ROS TF system follow the official tutorials first [http://wiki.ros.org/tf/Tutorials](http://wiki.ros.org/tf/Tutorials)). Now sample N points from the bird’s eye view and publish it as `visualization_msgs/MarkerArray`, so that you see the points on rviz (Remember to set the frame_id!). In order to convert the pixel coordinates of the output of `birdseye` to metric coordinates the following code-block might be useful.

```cpp
// C++ example
// Conversion from image coordinates to metric (meters) coordinates in the frame:
freicar_1/base_link
// pnt is a pixel coordinate of the output of the birdseye function
float x_m = pnt.y / 100.0f;
float y_m = pnt.x / 100.0f - reg_birds.cols/200.0f;
Eigen::Vector3f p_base_link(x_m, y_m, 0.0);
```
3 Submission

The submission is due on Dec 10, 2020, 23:59.
Submit your report as a PDF file and include a link to your repository. Your repository should include information of how to run your code!

Good luck!

References