Deep Learning deproducibility project

Hoaran Bi
H.Bi@student.tudelft.nl
5242525

Wouter de Leeuw
W.F.deLeeuw@student.tudelft.nl
4487753

Procedures

- Tried to get the existing project running (H&W)
- Data preprocessing (H: annotation and dataset; W: radar and image data fusion)
- Then in parallel, working on different parts of the model (H: FPN; W:Backbone)
- Created the CRFnet module that combined the previous building blocks (H&W)
- Implemented the loss functions (H)
- Implemented the training process (H&W)
- Implemented the NMS decoder (H)
- Created the mAP metric functionality (W)

(H: Haoran's task; W: Wouter's task)

Existing Project

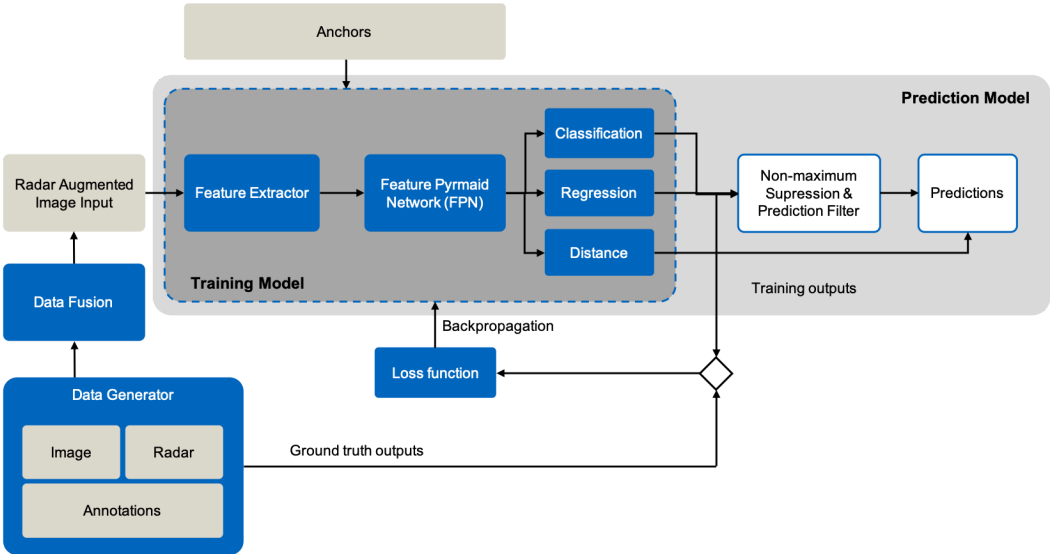
For this project we attempted to reproduce the paper '*A Deep Learning-based Radar and Camera Sensor Fusion Architecture for Object Detection*' by Nobos et al.
Our goal was not only to see if we could reproduce the results shown by this paper, but the external researcher also provided us with the challenge of altering the existing code, which was based on tensorflow, to make use of the Pytorch library.

The first step, after reading the paper, was to try to run the existing code, provided by the authors of the paper (https://github.com/TUMFTM/CameraRadarFusionNet). After facing driver issues using Linux, we ended up being able to run the code at least partially on Windows instead. This difficulty we had with getting the code to work, made us realize we had two ways to go about this project: either we could start with the existing code and alter it (in TensorFlow), or we could start from scratch (in PyTorch).

We found the existing project pretty complicated: consisting of a lot of scripts depending on each other. Besides, the existing code is adapted from another project, so many parts of the project are not necessarily needed in this project. So, we decided to start 'from scratch', while using the existing code as reference and possibly copying over a few useful functions. The author's code used the Keras implementation of RetinaNet, so, similarly, we used a PyTorch implementation of RetinaNet as our reference as well.

Our use of a notebook instead of using different python scripts also results in the added value of this project: working through the notebook makes the user go through the code in series using

clearly separated steps, instead of the parallel nature of using different scripts. This makes the code more accessible and easier to understand.



Project structure

Dataset

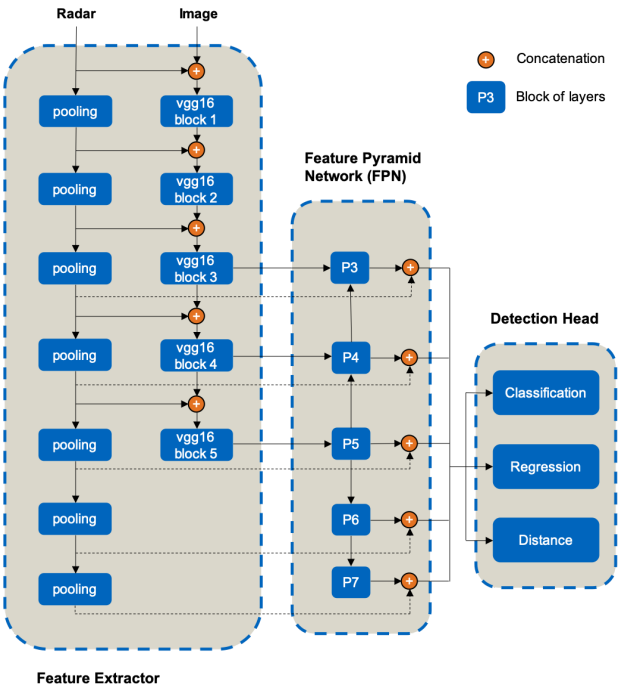
The project is based on the dataset NuScenes (https://www.nuscenes.org/nuscenes), which can be used for object detection tasks in the field of autonomous driving.

For our reproduction, we first started working on the preprocessing step. Although some of the preprocessing functions from the original code proved useful, we still needed to make our own custom Pytorch dataset class. Based on the NuScenes dev-kit and some existing preprocessing functions (mainly functions for radar and image fusion), we coded the 'get_item' and '_init_' function so that the class object would return an image (fused with corresponding radar data), and its annotations when traversing the dataset.

CRF Net

After setting up the custom dataset class, it was time for us to work on the network itself. The network consists of three main parts: the Feature Extractor, the Feature Pyramid Network (FPN) and the Detection Heads.

The Feature Extractor (or backbone) in the paper was based on the VGG16 network. The different blocks of the VGG16 network are applied to the radar and camera data separately, and after every block the data is concatenated again. Pytorch does have the option to import the VGG16 network, so we initially decided to create a Pytorch model based on this existing one, but including the data fusion.



However, a persistent error when importing the premade VGG16 network made us decide to code it ourselves instead. The created module outputs a dictionary containing the 8 different outputs that feed into the FPN.

The Feature Pyramid Network (FPN) is used here to generate a rich feature pyramid with multiple scales. The outputs of the FPN network are then further fused (concatenated) with the radar data and fed to the classification head and regression head respectively. The outputs of both heads are then concatenated together as the final outputs or CRF net.

After creating these different modules, we combined them together in the 'CRFNet' class. In addition, we generate anchors for all features output from the FPN.

Training and testing

Before training, we also implemented custom loss functions for classification output (focal loss) and regression output (smooth L1 loss). The sum of these two losses will be later used as the loss for backpropagation.

To train the network the custom dataset is loaded using a dataloader, and iterated through. This is then repeated for a specified amount of epochs. Every epoch outputs two lists: one of detections, and one of annotations. Every item in these lists refers to a specific sample, and contains a tensor with the corresponding detections/annotations.

After the model is trained, we implement a decoder to transform the output of detection heads to prediction boxes. This can be done using non-maximum suppression (NMS). NMS receives the proposal boxes (regression head outputs and anchors) and confidence score (classification head outputs) as input, output filtered proposal boxes with label (prediction box). The filter is based on two criteria: first, the box needs a high confidence score. Second, the box cannot have a large overlap area with other boxes having the same label.

The paper chooses to assess the performance of the network using the mean Average Precision (mAP). Since there was no premade function for this metric for us to implement, we had to create one ourselves.

We ended up splitting this task up in different functions. The 'precision_recall' function calculates the precision and recall for a certain iou value (amount of overlap between bounding boxes). This is done for every label for different iou values, to get the precision-recall curve. The area under this curve is the Average Precision (AP) for that label. Scaling the APs with the total instances of each label gives the mAP score.

Result

Because of the limitation of time and hardware, we were limited to only using the mini NuScenes instead of the full dataset. We followed the author's default configuration. The observed mAP score for the different epochs training the mini dataset fluctuate a lot, but did

show an upward trend. The fluctuations can be explained by the small size of the mini dataset, which can result in overfitting easily.

Although only trained and tested on the mini dataset, we can already see the positive trend of decreasing average loss and increasing mAP in the training processing.

In conclusion, we transformed the complicated existing project into its pytorch version, and presented it in a more readable way. We hope this work will help people understand the CRF network more quickly and reproduce the project more easily.