Special Topics - MSAI 495

Final Report

“On-road object detection for self-driving cars”

March 14th, 2022

Professor
Reda Al-Bahrani

Students
Ayushi Mishra
Preetham Pareddy
Ana Cheyre

Github link- https://github.com/amcheyre-nw/DL_Object_detection

Objective Description

Self driving cars might be a luxury in the current world but the technology is rapidly changing to be available for everyone. They are useful because ideally, machines don't make the mistakes humans do, so there would be less accidents like crashes. It will also save a lot of time for humans, making their life much more efficient. Computer vision is a critical component for the functioning of such cars. There is an absolute need to create better models that minimize, if not completely eradicate, danger on the road. This is the reason we chose to work on object detection on the road. The goal of this project is to develop a computer vision model that solves some of the multiple tasks that the system of self-driving cars must execute to be able to drive autonomously. This refers to the detection of diverse elements found on the streets so that the car is able to identify if this type of obstacle is on the road or not.

In this project, we are going to implement one of the best-known models in object detection which is YOLO. But we also want to see how this compares with another well recognized model called Faster R-CNN and thus see which of the two gives better results.

Related Work

Object Detection has always been a burning issue in computer vision which is applied in many areas such as security, surveillance, autonomous vehicle systems, and machine inspection. The popular object detector algorithms implemented in all these domains are either region-based detection algorithms or single-shot detection algorithms.

Baseline

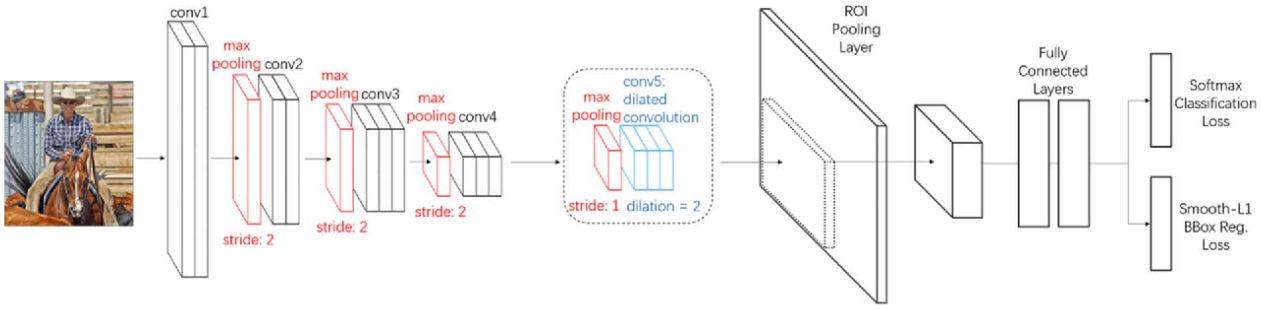
The paper that strongly influences our project works and forms a baseline for us is the R-CNN (Region based Convolutional Neural Network) family of papers by a group of researchers at UC Berkeley (Ross Girshick et al.). In the R-CNN family of papers that we are referring to, the evolution between different versions mentioned in the papers was usually in terms of computational efficiency (incorporating the various stages of training), decreasing testing time, and improving the model performance (mAP).

- **R-CNN (Ross Girshick et. al, 2014)** In this paper, the issue of object detection is resolved by introducing a concept called R-CNN. The network comprises of:
    a. A region proposal algorithm that is responsible for generating “bounding boxes” that outlines the location of potential objects present within the image.

b. A feature generation stage that is responsible for obtaining features of these identified objects, with the help of a CNN.
c. A classification layer that is responsible for predicting the object's class.
d. A regression layer that is responsible for making more precise coordinates of the object bounding box.

- **Fast R-CNN (Ross Girshick, 2015)** Within a year of publication of the previous R-CNN paper the authors published this paper to build upon their previous work. Since R-CNN was slow, hard to train and had a large memory requirement; with fast R-CNN the authors combined the three different parts that we had in the R-CNN system (a CNN, SVM, Bounding Box Regressor) into one seamless architecture. *The fast R-CNN was found to train the VGG16 network 9 times faster than their previously developed R-CNN.* Moreover, it was found that the inference is 213 times faster and achieves a higher mAP. This introduced one seamless end-to-end system that could be trained with back-propagation.

- **Faster R-CNN (Shaoqing Ren et. al, 2022)** A Faster R-CNN object detection network is composed of a feature extraction network which usually is a pretrained CNN. This is then followed by two subnetworks which are trainable. The first is a Region Proposal Network (RPN), which is used to generate object proposals and the second is used to predict the actual class of the object. The primary differentiator for Faster R-CNN is the RPN which is inserted after the last convolutional layer. This is trained to produce region proposals directly without the need for any external mechanism like Selective Search. After this, ROI pooling is used and an upstream classifier and bounding box regressor.



Faster R-CNN architecture

Main approach

With Faster R-CNN as our baseline, we looked at other state-of-the-art approaches and found a paper YOLOv5 (Dong Wu et. al, 2022) In this paper, the authors introduce a novel yet simplistic and efficient network, that is capable of handling the three driving perception tasks of object detection, drivable area segmentation and lane detection simultaneously and have the ability to be trained end-to-end. Results indicate that the model performed exceptionally well and was either at par or beat the existing state-of-the-art methods for all three tasks. Moreover, it

pioneered real-time reasoning on embedded device Jetson TX2, thereby ensuring that the network can be translated well to real-world scenarios.

With 45 frames per second, YOLO establishes its superiority over other object detection algorithms by being orders of magnitude faster than them. The YOLO algorithm's only limitation is its inability to detect small objects within the image with great clarity due to spatial constraints. For e.g.- It may struggle in detecting a flock of birds.

Dataset and Pre-processing

Dataset

For our project we will be using the most diversified and popular open driving video dataset in the field of computer vision- the BDD100K. It contains around 120 million images of roads, with signs and different objects on them, obtained from 100,000 videos on roads where each video's specifications are as follows: 40 sec long, 720p, and 30 fps. The data was collected from various locations in the United States, and includes environmental factors like weather conditions and time of the day. Due to its diversity and resulting robustness, the algorithm trained on this dataset can be migrated with ease to a new environment. The database we are using already consists of training (70K images), validation (10K images) and test (20K images) datasets.

For object detection, each image has object bounding boxes, and these boxes contain the respective label. This aids in understanding object distribution and its respective location. The images contain labels for 10 different types of objects, which correspond to 10 different classes: traffic light, traffic sign, car, person, bus, truck, rider, bike, motor and train.

Class	Frequency
Traffic light	265,906
Traffic sign	343,777
Car	1,021,857
Person	129,262
Bus	16,505
Truck	42,963
Rider	6,461
Bike	10,229
Motor	4,296

Train	179

Frequency of classes in BDD100K database

The database is divided into images with a respective ID, and another folder with a JSON file for each dataset (train, validation, test) indicating the labels of the objects and other information of the image, like timestamp, scene description, time of day, coordinates of box location, etc.

Here we can see an example of an image and a fragment of its respective JSON. It has multiple labels, but in this case we are going to match the ID of the image with the field “name” of the JSON, then under the “labels” field, we can get the “category” of the object, which will indicate the class “car” in this case. Above that, we can get the 4 coordinates of the vertices of the box location of the object, corresponding to x1, y1, x2, y2.

Image ID: "0000f77c-6257be58.jpg”

JSON:


```
{
  "name": "0000f77c-6257be58.jpg",
  "attributes": {
    "weather": "clear",
    "timeofday": "daytime",
    "scene": "city street"
  },
  "timestamp": 10000,
  "labels": [
    {
      "id": "1",
      "attributes": {
        "occluded": false,
        "truncated": false,
        "trafficLightColor": "NA"
      },
      "category": "car",
      "box2d": {
        "x1": 49.44476737704903,
        "y1": 254.530367,
        "x2": 357.805838,
```

```
        "y2": 487.906215
      }
    }, (...)
  ]
}
```


Pre-processing

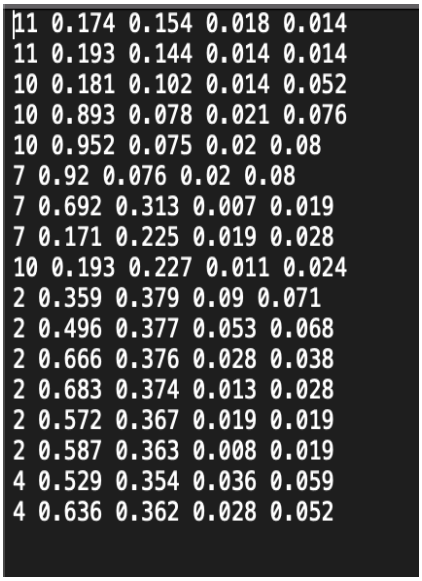
To improve the performance of our model, we will need to preprocess the data using different methods.

First, we have to process the images files, both resizing and converting them into pixel arrays in order to be able to work with them in the models. The input size of the images is 1280×720×3, and we will resize them to 640×384×3 that will allow the model to run faster, but still keep all the information needed for model training. Then, to convert the images to pixel arrays, we'll use the *Pillow* library, which takes an image and returns the content as an object containing pixel values, which can be transformed into a numpy array. Each array contains 3 channels, which represent the colors red, green and blue from "RGB" color code, and the value of each pixel between 0 and 255 with respect to each color channel.

Second, we will implement data augmentation to further increase image variability, thereby improving the robustness of our model within various environments. Photometric and geometric distortions will be used to create more data. For photometric distortions, we will play with the image HSV (hue, saturation, value). On the other hand, for handling geometric distortions we plan to perform some of the affine transformations. To carry out these transformations, the *TorchVision* library will be used, which has integrated functions that allow these tasks to be carried out straightforwardly.

Preprocessing for YOLO

The process that took most of our time was the preprocessing part. YOLO requires the input label files to be of a certain format. So we had to first change the BDD100 format to COCO and then change that to YOLO format. In the below images, the left one is the json of the original BDD100k dataset. It contains the labels of all the objects in the image. The right image is the YOLO format which contains image class and the bounding box coordinates for one section of an image. Each image is divided into multiple sections (for example 9) and the object detection is performed on each of those sections and then aggregated to get the objects in the image.



The labels of the dataset are : person,rider,car,bus, truck, bike , motor ,tl_green , tl_red, Tl_yellow, tl_none, traffic sign, train , tl_green.

Methodology

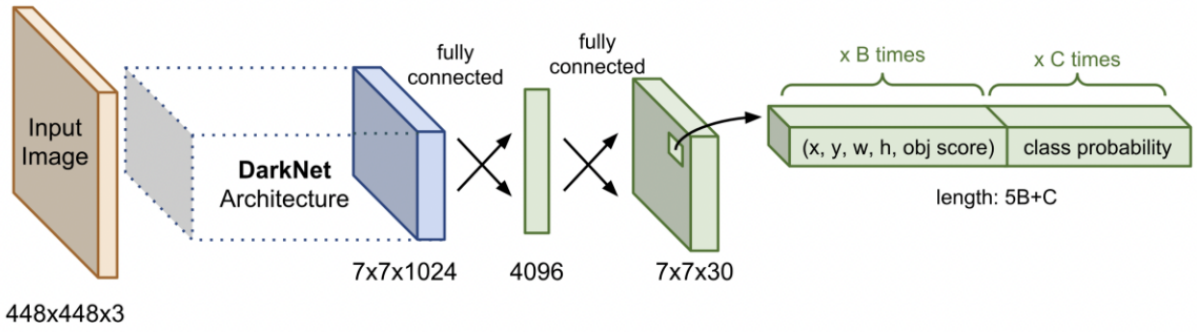
The main algorithm which we try to copy for this purpose is YOLOv5 since it's considered the state of art for the task of object detection on the road. Because of the data shared by multitask, the prediction results of YOLOv5 are more reasonable. For instance, YOLOv5 won't misidentify objects removed from the road as vehicles. Moreover, the samples of false negatives are much less and also the bounding boxes are more accurate. However, this is dependent on the computing power that's available to us.

Architecture

YOLOv5 is composed of one encoder for feature extraction and a decoder to handle the object detection. More specifics about each part are given below:

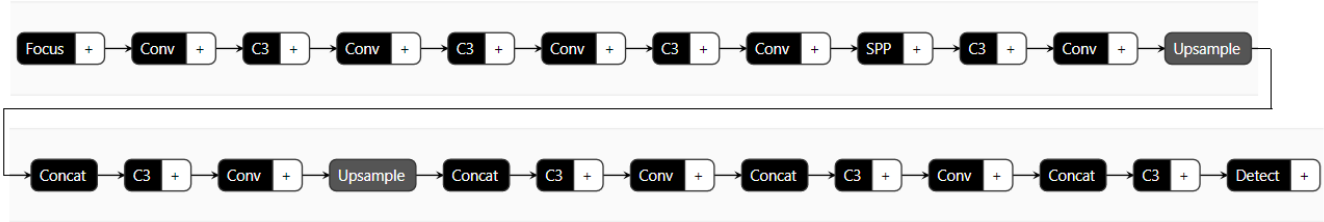
Encoder - The encoder contains a backbone and a neck. The backbone performs the task of extracting features from the image. CSPDarknet is being used for this purpose. The neck is employed to fuse the features generated by the backbone. Our neck is principally composed of the Spatial Pyramid Pooling (SPP) module and therefore the Feature Pyramid Network module.

Decoder: According to Dong Wu et al. (2022) an anchor based multi-scale detection scheme is being employed as the framework for object detection. Initially, Path Aggregation Network (PAN), a bottom-up feature pyramid network, was used. FPN is responsible for transferring semantic features top-down, and PAN is responsible for transferring positioning features bottom-up. Both are then combined to get a higher feature fusion effect, so the multi-scale fusion feature maps within the PAN are used for detection. Then, each grid of the multi-scale feature map is going to be assigned three prior anchors with different aspect ratios, and also the detection head will predict the offset of position and also the scaling of the peak and width, similarly because of the corresponding probability of every category and also the confidence of the prediction.



YOLO architecture

The darknet architecture contains 24 convolutional layers as shown in the image below, which forms the backbone. The neck contains the fully connected layers to classify the features extracted from the backbone. The head is what contains the output, whichever format is desired. The darknet is shown below-



Darknet architecture

Loss

Dong Wu et al. (2022) further explains how the model is tasked with performing three tasks with the help of three decoder, the loss function consists of three components as shown below:

$$\mathcal{L}_{det} = \alpha_1 \mathcal{L}_{class} + \alpha_2 \mathcal{L}_{obj} + \alpha_3 \mathcal{L}_{box}$$

where

L_{class} and L_{obj} : focal loss, which is utilized to reduce the loss of well-classified examples, thus forcing the network to focus on the hard ones.
L_{class} - penalizing classification
L_{obj} - confidence of one prediction.
L_{box} is L_{CIoU} - takes distance, overlap rate, the similarity of scale and aspect ratio between the predicted box and ground truth into consideration.

The metrics used for the object detection task would be:

We will be measuring mainly the mean average precision (maP) and the mean average recall (maR), which are the most common metrics in object recognition as it measures how good the model is at performing the classification of objects. Finally, we will keep track of the training loss after each iteration and computation time. Additionally, other object detection specific evaluation metrics we will consider are multiple object tracking precision, IoU (intersection over union), multiple object tracking accuracy, and mean average precision with heading.

Like we have mentioned before, we are focusing on recall which is a common norm in traffic object detection. The top models for this dataset are:

YOLOv5	89.2
YOLOv5s	86.8
Multinet	81.3
Faster R-CNN	77.2

We will measure the performance of our neural network by measuring it against the above existing models. As a baseline, we will attempt to outperform Faster R-CNN. Recent state of the art models have improved this score by at least 10% Thus, we will attempt to reach a recall of 87%.

Implementation

For the development of the models, we used python for programming the code, and several libraries related to deep learning modeling, among them the most important PyTorch. For the execution of the code, we chose to do it in Colab Pro, which provides GPU P100; with 25 GB of RAM, and 226 GB of disk space.

For a fair comparison between both models, the same 20,000 images were used for training and 2,000 for testing.

Faster R-CNN

For the implementation of the Faster R-CNN model we used a PyTorch module that has a pre-trained model with a ResNet-50-FPN backbone. The model was pre-trained in the COCO database, which has 80 classes of different objects, including the 10 specific ones needed for self-driving cars. The goal was to fine-tune the model to the BDD100K database and compare the results obtained with the YOLO model.

Hyperparameters used:

- Trainable backbone layers = 3
- Image output size = (432, 768)
- Batch size = 16
- Epochs = 10
- Learning rate = 0.01 (incremental)
- Training set = 20,000 images
- Testing set = 2,000 images

Epoch	Loss	maP	maR	Run time
0	1.2818	0.305	0.396	0:34:29
1	1.3624	0.365	0.472	0:29:29
2	1.3366	0.373	0.479	0:29:37
3	1.3599	0.384	0.476	0:29:40
4	1.2748	0.382	0.479	0:29:45
5	1.3177	0.383	0.482	0:29:45
6	1.3078	0.383	0.481	0:29:39

7	1.2608	0.383	0.481	0:29:38
8	1.3001	0.383	0.482	0:29:35
9	1.3275	0.383	0.481	0:29:37

Results of Faster R-CNN model

YOLO v5

For this model we used the pretrained model from the ultralytics code base. The model had been pre trained on COCO data with 80 classes. We used pytorch to fine tune this model on the BDD100K dataset.

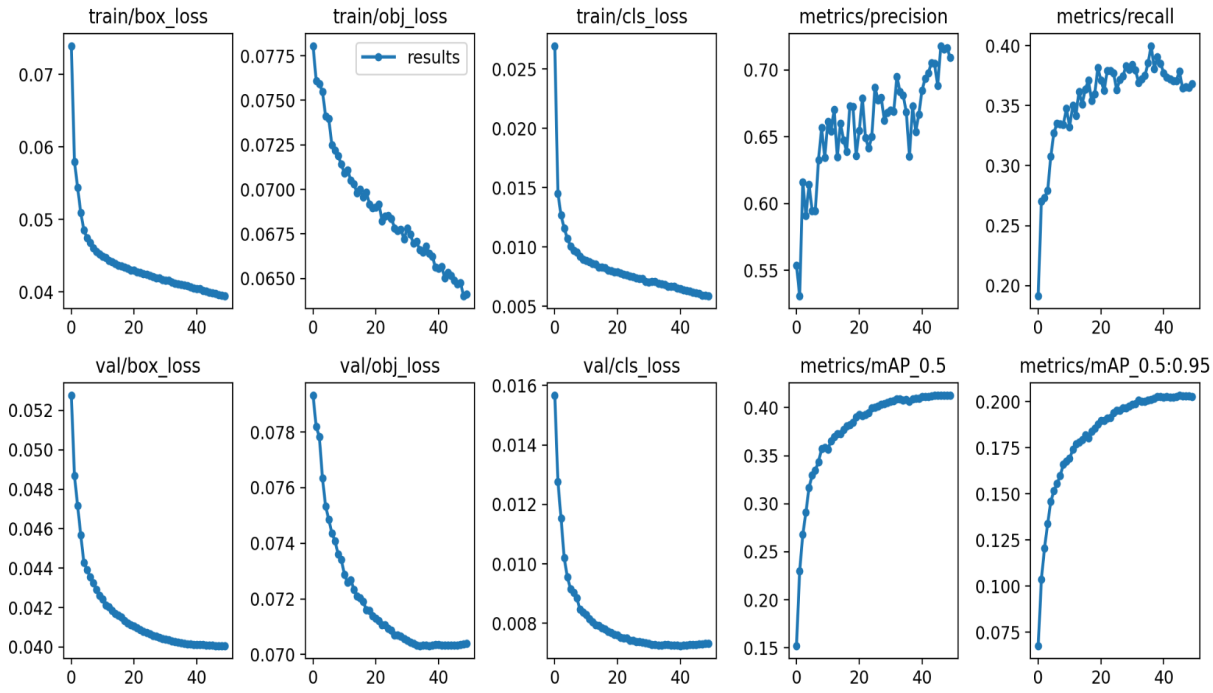
Training Info:

- Trainable backbone layers = 3
- Image output size = (432, 768)
- Batch size = 16
- Epochs = 10
- Training set = 20,000 images
- Testing set = 2,000 images

Hyperparameters used:

lr0	0.01
lrf	0.01
momentum	0.937
weight_decay	3.0
warmup_epochs	0.8
warmup_momentum	0.1
box	0.05

The YOLOv5s was run for 50 epochs and the results are as follows:

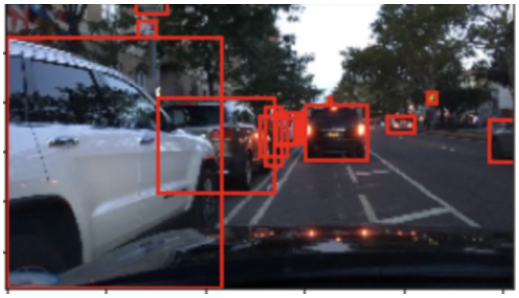


Analysis

Theoretically as well as based on our results we can see that YOLO v5 and Faster RCNN both share some similarities. They both use an anchor box based network structure along with bounding box regression. Something that differs YOLO v5 from Faster RCNN is that it makes classification and bounding box regression at the same time which based on our research outlined in related work makes sense (YOLO v5 was released later and is therefore more elegant).

The YOLO v5 model seems much better at detecting smaller objects — traffic lights in this case and also is able to pick up the car when it is farther away i.e smaller. Little to no overlapping of boxes is there for the YOLO v5 model when compared to the faster RCNN model. Both these models struggle to detect objects in the distance especially when the image is dark i.e. has low light and the objects in the image are smaller. Since we took batches of our dataset it is possible that in our runs the batch we choose at random had more dark images thereby rendering the maP and maR values low.

Sample output image that our models can't predict

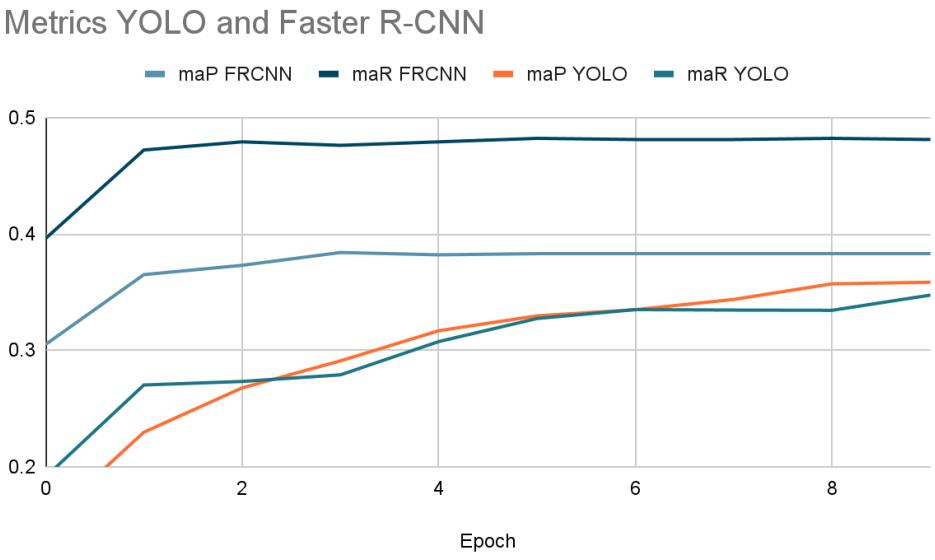


Sample output image that our models can predict

Moreover, YOLO v5 has a clear advantage in terms of run speed. The YOLO v5 model runs about 2.5 times faster while managing better performance in detecting smaller objects. The results are also cleaner with little to no overlapping boxes. Since we are using images of the video and not the video itself, the known advantage of YOLO in real time over faster RCNN in this respect cannot be talked about in scope of this project.

	maP	**maR**	**Runtime**	**GPU**
YOLO v5	0.44	0.38	4hrs	K80
Faster R-CNN	0.38	0.48	5 hrs	P100

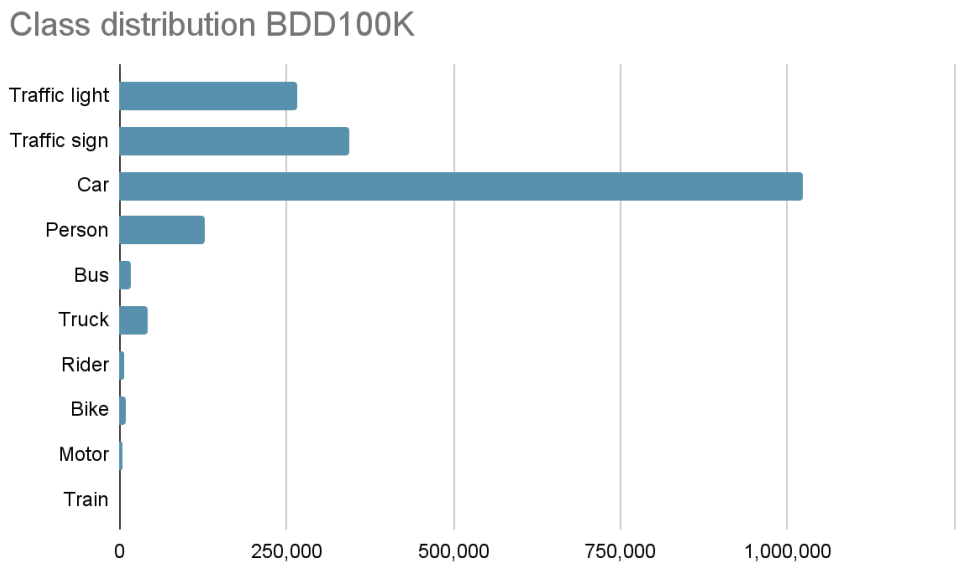
Final results for YOLO v5 and Faster R-CNN models



Metrics comparison between YOLOv5 and Faster R-CNN

Based on the metrics comparison done above using graphs and results above we can conclude that-

- The start of Faster R-CNN itself is pretty high and then it peaked early, whereas YOLO started learning at a constant rate. But this is the first 10 epochs. YOLO plateaus in the next few epochs. But from this it is clear that faster R-CNN started from a high mAP which might mean the weights of the pretrained model are better for this dataset.
- YOLO performed the same with twice the amount of data (10k vs 20k). It was interesting to note that doubling the number of images fed to the models, did not translate to an increase in maP or maR, rather it was maintained. One theory is that the data in this **database is very unbalanced**, so the features we are adding are too noisy or do not have enough variation to capture critical patterns in the data, therefore increasing the data was useless regardless of the volume of data.



Improvements and future work

The model can still be improved. We tried increasing the size of the images and increasing the number of images for training. We still have left:

- Increasing the number of images of underrepresented classes.
- Increasing the number of convolutional layers, for example using a ResNet128 instead of ResNet50.
- Another option to explore is to use other weights for the pre-trained models and see if they perform better.
- Computational issues could be solved using AWS.

Reference list

Ross Girshick, Jeff Donahue, Trevor Darrell, Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. arXiv:1311.2524 [cs.CV](2014)

Ross Girshick. Fast R-CNN. arXiv:1504.08083 [cs.CV] (2015)

Shaoqing Ren, Kaiming He, Ross Girshick, Jian Sun. Faster R-CNN: Towards Real-Time Object Detection with Regional Proposal Networks. arXiv: 1506.01497 [cs.CV] (2016)

Dong Wu, Manwen Liao, Weitian Zhang, Xinggang Wang. YOLOv5: You Only Look Once for Panoptic Driving Perception. arXiv:2108.11250v6 [cs.CV] (2022)