Special Topics - MSAI 495

Project Progress – Update Report

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

February 28th, 2022

Professor
Reda Al-Bahrani

Students
Ayushi Mishra
Preetham Pareddy
Ana Cheyre

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/Literature Survey

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)** The Faster R-CNN architecture consists of the RPN(another convolutional network) as a region proposal algorithm and the Fast R-CNN as a detector network. This resulted in decreasing the region proposal time per image (2s to 10ms) but also allowed the region proposal stage to share layers with the following detection stages, resulting in an overall significant improvement of feature representation.

Main approach

With Faster R-CNN as our baseline, we looked at other state-of-the-art approaches and found a paper YOLOP (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.

Project extension

We will closely follow the approaches mentioned in the above papers. However when it comes to our main approach we will tweak the state of the art methods by using different methods like warmup and cosine annealing on the learning rate of the model to create effective learning rate schedules. We also hope to explore the possibility of integrating the R-CNN modeling techniques with YOLO.

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:

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.

Methodology

The main algorithm which we try to copy for this purpose is YOLOP since it's considered the state of art for the task of object detection on the road. It's a panoptic driving perception network to perform traffic object detection, drivable, area segmentation and lane detection simultaneously. Though our main goal is object detection, supporting the research we've got done, the opposite tasks (drivable area segmentation and lane detection) help in improving the performance of object detection. Because of the data shared by multitask, the prediction results of YOLOP are more reasonable. For instance, YOLOP 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. So our main task is object detection but the drivable area segmentation and lane detection tasks are optional.

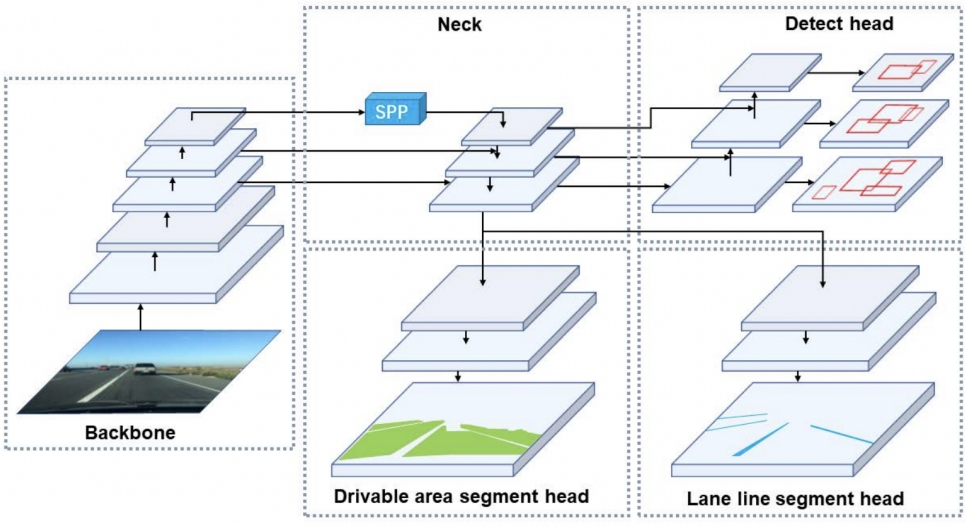
Architecture

YOLOP is composed of one encoder for feature extraction and three decoders to handle the specific tasks. 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 - The decoder consists of three heads for the three specific tasks the model is getting used for i.e traffic object detection, drivable, area segmentation and lane detection.

- ***Detect head*:** 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 are 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.
- ***Drivable Area Segment Head & Lane Line Segment Head (optional)* :** Dong Wu et al. (2022) further states that both the processes adopt the identical network structure.The bottom layer of feature pyramid network is fed to the segmentation branch, with the dimensions of (W/8, H/8, 256). After three upsampling processes,the output feature map is restored to the dimensions of (W, H, 2), which represents the probability of every pixel within the input image for the drivable area/lane line and therefore the background. due to the shared SPP within the neck network, no extra SPP module is added to segment branches. Additionally, the closest Interpolation method is employed within the upsampling layer to scale back computation cost rather than deconvolution. As a result, not only do the segment decoders gain high precision output, but even be in no time during inference.



YOLOP 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.

Training structure

Input: Target neural network F with parameter group:
Θ = {θenc, θdet, θseg}
Training set: T
Threshold for convergence: thr
Loss function: L_{all}
Output: Well-trained network: F(x; Θ)

1: procedure TRAIN(F, T)
2: repeat
3: Sample a mini-batch (xs, ys) from training set T .
4: ` ← Lall(F(xs; Θ), ys)
5: Θ ← arg minΘ `
6: until ` < thr
7: end procedure
8: Θ ← Θ\{θseg} // Freeze parameters of two Segmentation heads.
9: TRAIN(F, T)
10: Θ ← Θ ∪ {θseg} \ {θdet, θenc} // Freeze parameters of Encoder and Detect head and activate parameters of two Segmentation heads.
11: TRAIN(F, T)
12: Θ ← Θ ∪ {θdet, θenc} // Activate all parameters of the neural network.
13: TRAIN(F, T)
14: return Trained network F(x; Θ)

Though the model is state of art we want to tweak it in certain ways for experimentation. We want to try different methods like warmup and cosine annealing on the learning rate of the model to create effective learning rate schedules. Apart from this we want to research R-CNN further to see if we can incorporate some techniques used in that model with the YOLOP.

The metrics used for the object detection task would be:

Precision (P) and recall (R) for each object category, in this case we will be particularly interested in the amount of false negatives (looking for high recall), because in a self-driving car there is no space for errors because it can risk people's lives. Also the mean average precision (MAP) which is a very common metric in object recognition as it measures how good the model is at performing the classification of objects. Finally, we will keep track of accuracy and the training loss after each iteration.

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:

YOLOP	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%. 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.

We intend to investigate, and potential apply the following methods to aid in improving accuracy and precision:

- ***Implementing different sampling strategies:*** Since point clouds are usually sparse, by sampling we would be able to improve the data imputed to the NN.
- ***Weighted boxes fusion:*** From various object detection models ensembling boxes.
- ***Restricting our training dataset to high visibility and minimal occlusion frames:*** Using frames without occlusions/better visibility and good bounding box info can improve training. Using subsampling, we can choose the frames that give the best visibility with respect to the bounding box.

Computational Problem

Since we do not have a lot of time and computational power to train, we want our architecture to perform well with less training. Therefore, we will explore methods expected to give us better inference. We plan to start with existing state of the art architectures, then propose a novel model by modifying these models.

Prototype

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 K80, P100, T4; 25 GB of RAM, and 226 GB of disk space.

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 91 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. To start with the development of the model, for the moment only 200 images have been used for the training.

One of the biggest challenges has been minimizing the time it takes to run the model and having enough RAM to run it. To address this, a number of things have been attempted:

- **Batch size increase:** Increasing the batch size will lead to a decrease in the number of batches to process in the training, which should take less time. But we found that when doing this, the necessary amount of RAM for each batch increased, leaving the program without the necessary capacity, crashing the model.
- **Freeze weights**: One of the tasks that take most of the time and computation is the updating of the weights after the backpropagation of the neural network. For this reason, we decided to freeze the weights that the pre-trained model already has, with “param.requires_grad = False”. Although we believe that something could have improved in computation, it is not yet seen in a very noticeable change.
- **Resnet18**: To reduce the amount of computation, we searched for an existing Faster RCNN model that uses Resnet18 instead of Resnet50, but unfortunately the smallest existing one is with Resnet50.

Epoch	Batch group	Loss	maP	Run time
0	1	9.1	0.000	2:01:28
	2	8.6	0.000	
1	1	8.1	0.000	2:00:33
	2	8.0	0.001	

Preliminary results of Faster R-CNN model

The current results obtained with the model are still not good. Given the amount of time the model takes to run it has been difficult to debug, and for training so far we have only been able to run up to 2 epochs. Although we see that the loss in each iteration becomes smaller, which means that the model is learning, the maP we see at the moment is 0 in each iteration of the training. One theory about this is that there may be an incompatibility or some associated error when comparing the classes of the detected objects, always reaching a mismatch. In addition to the above, to achieve better results it is necessary to train the model with a greater amount of data and with more epochs, which until now has not been possible due to execution time problems.

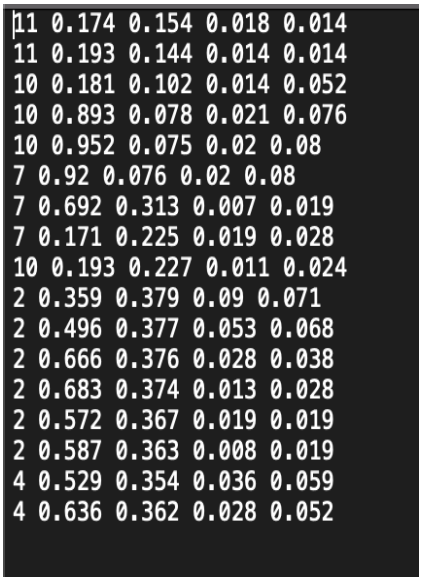
YOLO v5

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.

Preprocessing

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 left 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.

```
[
    {
        "name": "b1c66a42-6f7d68ca.jpg",
        "attributes": {
            "weather": "overcast",
            "scene": "city street",
            "timeofday": "daytime"
        },
        "timestamp": 10000,
        "labels": [
            {
                "category": "traffic sign",
                "attributes": {
                    "occluded": false,
                    "truncated": false,
                    "trafficLightColor": "none"
                },
                "manualShape": true,
                "manualAttributes": true,
                "box2d": {
                    "x1": 1000.698742,
                    "y1": 281.992415,
                    "x2": 1040.626872,
                    "y2": 326.91156
                },
                "id": 0
            },
            {
                "category": "traffic sign",
                "attributes": {
                    "occluded": false,
                    "truncated": false,
                    "trafficLightColor": "none"
                },
                "manualShape": true,
                "manualAttributes": true,
                "box2d": {
                    "x1": 214.613695,
                    "y1": 172.190058,
                    "x2": 274.505889,
                    "y2": 229.586743
                },
                "id": 1
            },
            {
                "category": "traffic sign",
                "attributes": {
                    "occluded": false,
                    "truncated": false,
                    "trafficLightColor": "none"
                },
                "manualShape": true,
                "manualAttributes": true,
                "box2d": {
                    "x1": 797.314833,
                    "y1": 313.186265,
                    "x2": 829.756437,
                    "y2": 341.884608
                },
                "id": 2
```

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.

Model

The model was run on 5800 images as training and 194 images as validation. This is not optimal but since moving all these images in google drive was a very slow and tedious task and because of the time constraints we are presenting the results for these images. Some of the hyperparameters of the models are as follows:

Learning Rate	0.01
Momentum	0.937
Weight Decay	0.0005
Warm_up momentum	0.8

The architecture of the model is given in the image below.

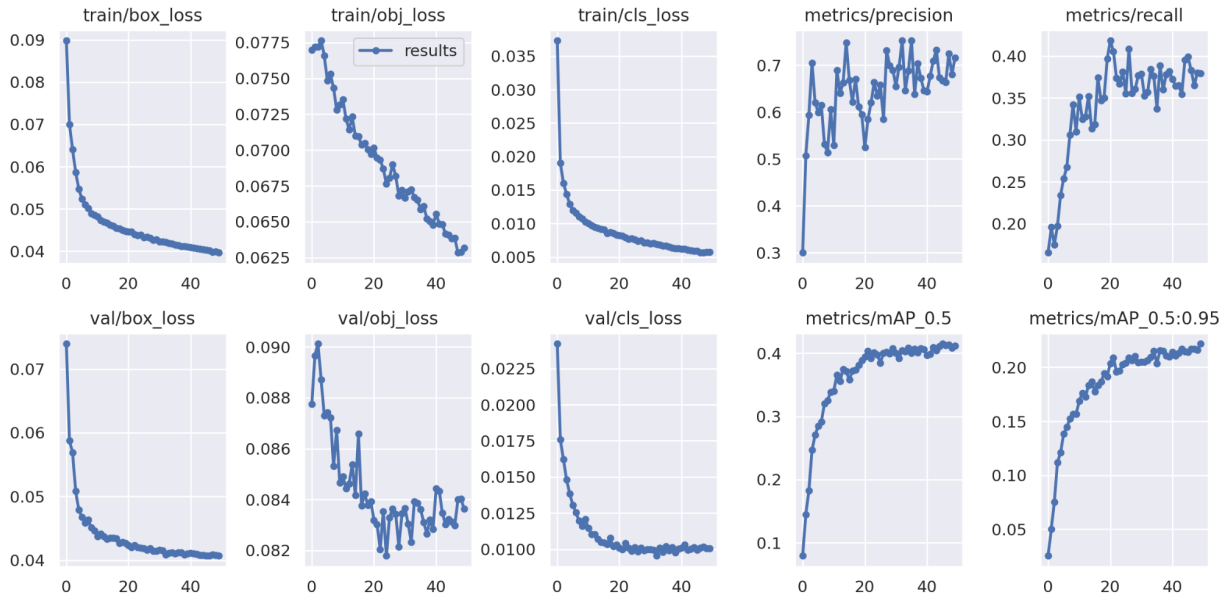
```
                from  n    params  module                                  arguments
  0                -1  1      3520  models.common.Conv                      [3, 32, 6, 2, 2]
  1                -1  1     18560  models.common.Conv                      [32, 64, 3, 2]
  2                -1  1     18816  models.common.C3                        [64, 64, 1]
  3                -1  1     73984  models.common.Conv                      [64, 128, 3, 2]
  4                -1  2    115712  models.common.C3                        [128, 128, 2]
  5                -1  1    295424  models.common.Conv                      [128, 256, 3, 2]
  6                -1  3    625152  models.common.C3                        [256, 256, 3]
  7                -1  1   1180672  models.common.Conv                      [256, 512, 3, 2]
  8                -1  1   1182720  models.common.C3                        [512, 512, 1]
  9                -1  1    656896  models.common.SPPF                      [512, 512, 5]
 10                -1  1    131584  models.common.Conv                      [512, 256, 1, 1]
 11                -1  1         0  torch.nn.modules.upsampling.Upsample    [None, 2, 'nearest']
 12          [-1, 6]  1         0  models.common.Concat                    [1]
 13                -1  1    361984  models.common.C3                        [512, 256, 1, False]
 14                -1  1     33024  models.common.Conv                      [256, 128, 1, 1]
 15                -1  1         0  torch.nn.modules.upsampling.Upsample    [None, 2, 'nearest']
 16          [-1, 4]  1         0  models.common.Concat                    [1]
 17                -1  1     90880  models.common.C3                        [256, 128, 1, False]
 18                -1  1    147712  models.common.Conv                      [128, 128, 3, 2]
 19         [-1, 14]  1         0  models.common.Concat                    [1]
 20                -1  1    296448  models.common.C3                        [256, 256, 1, False]
 21                -1  1    590336  models.common.Conv                      [256, 256, 3, 2]
 22         [-1, 10]  1         0  models.common.Concat                    [1]
 23                -1  1   1182720  models.common.C3                        [512, 512, 1, False]
 24      [17, 20, 23]  1     51243  models.yolo.Detect                      [14, [[10, 13, 16, 30, 33, 23], [30, 61,
Model Summary: 270 layers, 7057387 parameters, 7057387 gradients, 16.0 GFLOPs
```

YOLO seems to be running much faster than the Faster RCNN. We shall compare the results when we are able to train the models in a much better manner.

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

precision	0.71
recall	0.38

mAP	0.411



Based on this we can see that the model is not performing that well. We can increase the epochs and slowly increase the amount of data being fed into the model to get better results.

Remaining work

Faster R-CNN

In order to achieve better results, there are still things to try, so we made a list of different actions that will help us to be able to reach our goal:

- **Simplify architecture:** Currently our model uses many pre-existing functions and modules that should make it easier to implement Faster R-CNN, but on the contrary, it is making debugging more difficult. It may also have additional functions that are not necessary for our model, but that increases the computation time even more. This is why one of the actions we want to do is try to build the same model but with more basic and visible functions, which will allow us to more easily identify what is preventing the correct execution of the model. We will focus mainly on the functions “train_one_epoch” and “evaluation”, which are the ones that are taking the longest of all.

- **Improve memory and RAM usage**: There are many ways to maximize the performance of a model, some of them are:

- Enable pinned_memory. This automatically puts the fetched data tensors in pinned memory and enables faster data transfer to CUDA-enabled GPUs.
- Use 16-bit precision. In 16-bit training parts of the model and the data go from 32-bit numbers to 16-bit numbers. This uses less memory and speeds up computation.
- Profile the code with *cProfile.* It gives a set of statistics that describes how often and for how long various parts of the program are executed. This can give an idea of which are the slowest functions and the most ones used.

We want to implement some of them to be able to make the execution more efficient.

Then, once the model is well built and tested with a small database, we want to increase the amount of data that is fed to the model training in order to achieve a maP and recall as high as possible. The size of the database will depend on Colab's ability to successfully process the information.

Finally, we are going to measure the same metrics as the YOLO model in order to make the corresponding comparison and understand which of the two models is the best to perform the object detection.

YOLO v5

Similar to the Faster RCNN we want to use the same techniques to improve the memory usage as we want to work on much more data for a larger number of epochs.

Better Results:

As of now we used 5000 training images and 194 validation images, We have more than 100,000 images with us and want to use these to slowly increase the training and validation size. We want to increase the number to 20,000 so that the model can perform better. Also, the number of epochs can be increased. We are planning to take a balanced sample of the 20,000 so we will loop through each json file and get the classes, that way we can make sure to keep balanced classes.

Hyperparameter Tuning:

There are a lot of hyperparameters that can be tuned like warmup, lrf, weight decay, optimizer etc. Though it would be computationally intensive, we want to try our best to experiment more with the hyperparameters.

Additionally, we want to try and modify the architecture of the model to get a much deeper understanding of how it is working. However, additional research will have to be done on this.

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. YOLOP: You Only Look Once for Panoptic Driving Perception. arXiv:2108.11250v6 [cs.CV] (2022)