# Selection of Problem Statement

I have selected the first problem statement for the Software Challenge. This is because I feel that there is a need to document and demonstrate the need for the best fitting algorithms in order to contribute positively to my future projects in this field.

Since this is an IITI competition, I will be writing this file in a way that is relevant to my future projects and competitions for which I have registered through the institute. I will detail my usage, approach, references and overall process to reach the final result.

# Selection of Vehicle tracking algorithm

I will include some relevant research papers that I read while trying to tackle this problem statement. My initial approach was to document MAP(Mean Average Precision) and FPS(Frames per second) of a lot of relevant algorithms by scaling their performance with respect to each other keeping one algorithm as a baseline. I set values of Faster RCNNs as 1 and tried to document relative values of other popular algorithms.

I quickly realized that the problem was a lot more complex. Firstly, different parameters and constraints make certain models preferable over others in certain conditions. For example, we might require higher precision in some scenarios while we may prioritize speed in others. Further, AP values of algorithms can vary when the objects to be detected are smaller or larger in size(shown later).

Note that precision doesn’t equate to MAP even relatively [[1]](https://towardsdatascience.com/map-mean-average-precision-might-confuse-you-5956f1bfa9e2). This example[[2]](https://assets.researchsquare.com/files/rs-668895/v1_covered.pdf?c=1631875157#page=19) shows how despite having higher mAP values, Faster CNNs have lower precision than YOLOv3 and SSD algorithms. Even if I were to take all of the relevant values, I found that algorithms will perform with different relative speed and accuracy when given different datasets and run on different hardware[Will be shown in great depth for YOLOv4 vs YOLOv5 comparison].

Thus, this approach was not a fair comparison for these systems even under the constraints provided by the problem statement. However, I do feel that some research in this area is needed for the field of computer vision as a whole and a better method than AP for comparison.

Clearly, I had to make my search more relevant and practical for autonomous driving and look at what my system needs from any algorithm. Quick decision-making is essential for cars of any level of autonomy so the first constraint I put was time.

RetinaNet stuck out immediately as some of the best algorithms in terms of AP at all levels.

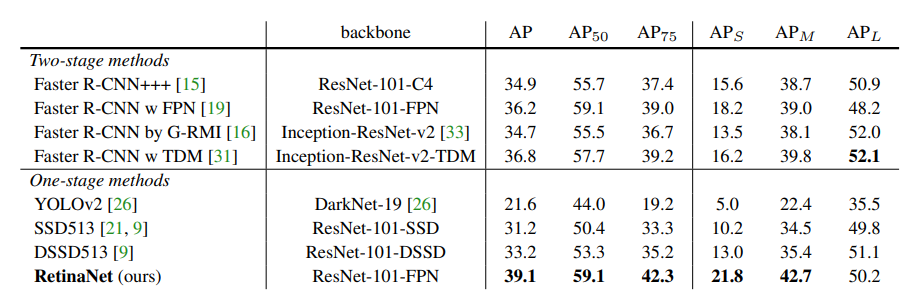


Table taken from RetinaNet’s paper[[3]](https://openaccess.thecvf.com/content_ICCV_2017/papers/Lin_Focal_Loss_for_ICCV_2017_paper.pdf) . S, M and L showing performance for small, medium and large objects respectively. Based on the COCO dataset

HOWEVER, the algorithm runs at around 3 frames per second. So I got curious if future iterations of YOLO are better and I found a YOLOv3 paper that cites this one [[4]](https://pjreddie.com/media/files/papers/YOLOv3.pdf). Page 3 and 6 of this paper are of great relevance since we cannot afford to get incorrect classifications for signals and road signs. It also emphasizes the speed of YOLO compared to other algorithms(even though page 4 and 6 has some pretty terrible attempts at data manipulation to make themselves look good in those graphs)

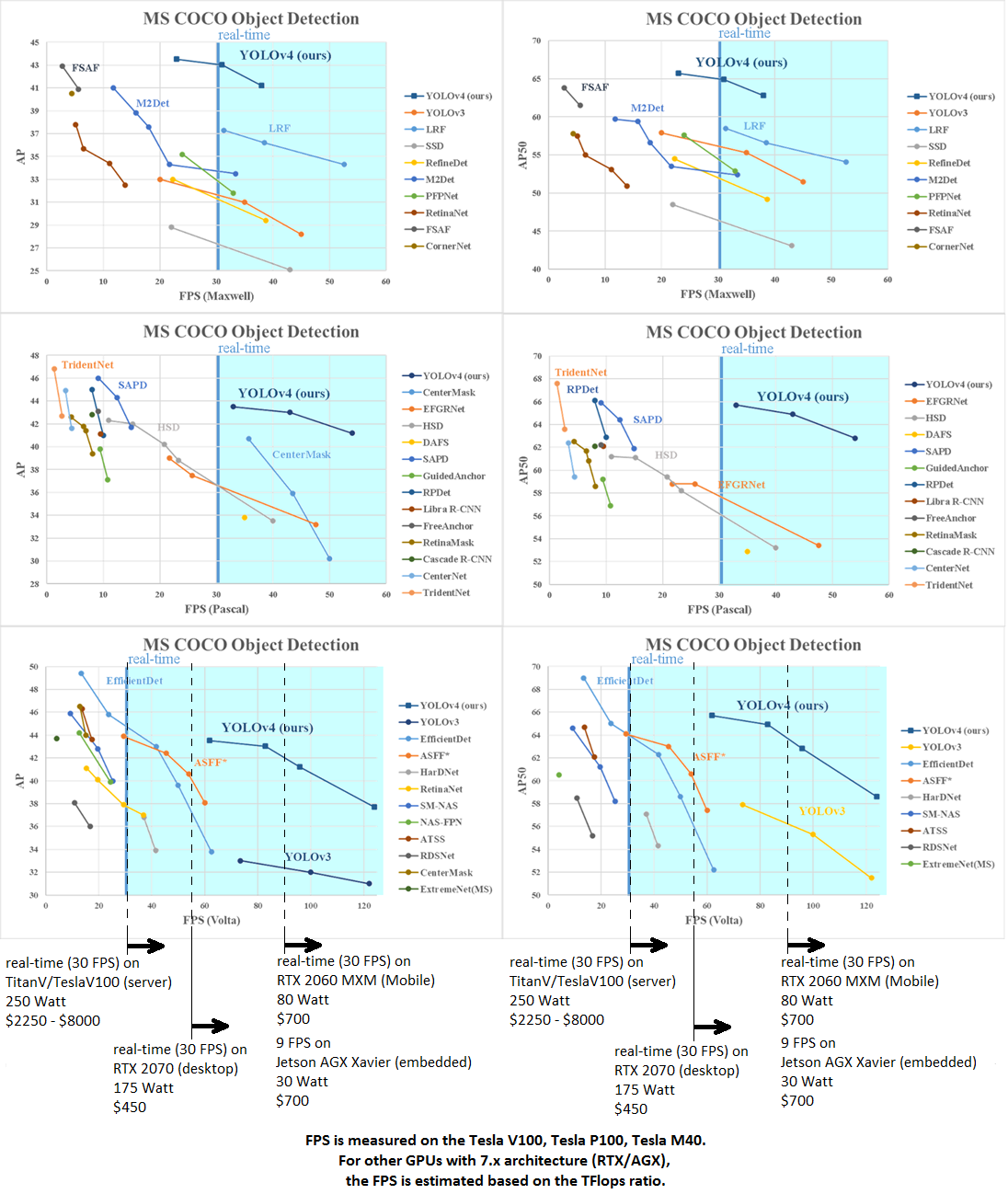
So I moved to YOLOv4 and YOLOv5. Firstly, YOLOv4 and YOLOv5 perform at an AP of 46 and 45 respectively at average latency of 9ms. This completely defeats any algorithm shown in the above sections.[[5]](https://blog.roboflow.com/yolov4-versus-yolov5/)

YOLOv4 has better documentation and has even provided its AP50 for the COCO dataset (65.7%) which is what YOLO seems to specialize in.

YOLOv5 has a much lower training time compared to YOLOv4 but has less customization and slightly less accuracy. YOLOv4 also has a lot of community support and is being heavily researched to this day so it is much more practical for our purposes.

I’m not mentioning everything I read but this article by the creator summarizes it well [[6]](https://alexeyab84.medium.com/yolov4-the-most-accurate-real-time-neural-network-on-ms-coco-dataset-73adfd3602fe)





A lot of algorithms are providing much more accurate models at lower FPS so as a final check, I looked for research papers with details on how much FPS is ideally needed for autonomous cars and found this paper. [[7]](https://digitalcommons.mtu.edu/cgi/viewcontent.cgi?article=1805&context=etdr) Their process is less sophisticated compared to RetinaNet or YOLO but that shouldn’t affect the fact that this is the optimum frame rate for the machine to make reliable decisions.

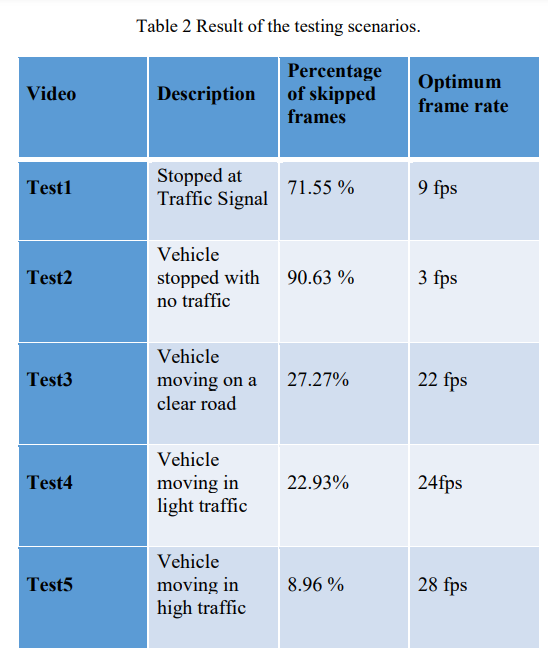


Table on page 25 of the attached research paper

The above paper is very interesting in its methodology and findings but for now let’s focus on the fact that a very high framerate is required. For more complex algorithms like the ones above, system integration might require them to have even higher values than the ones shown above. Also, my GPU is obviously not going to produce the highest possible FPS for a given algorithm so the fact that YOLO typically goes much higher also provides the required safety gap.

So it seems like YOLOv4 is designed perfectly for our use in this project with our **requirement being >22 fps**. For actual cars it might be useful to switch to other algorithms provided above since in stop position we don’t need a high FPS. Alternatively, we might employ the FPS reduction as suggested by the paper but again that is not relevant for the video provided to me.

[I later found this amazing website <https://paperswithcode.com/sota/object-detection-on-coco> . I will stick to YOLO series for now since I’m not sure if anything else will train in time for submission or if they even satisfy the FPS requirement]

[[[8]](%5b8%5d) was also a great read between YOLOv4, YOLOv5 and YOLO series and some mistakes in the original article]

Among the various improvements of YOLOv4, **YOLOR-D6 was the highest performing YOLO on the COCO dataset which contains car and bike images among various other relevant objects at 34 fps. It is also the 17th best-performing algorithm on the COCO dataset and the creator has shown through multiple threads, papers and conversations that it is practically the best algorithm for 30+ FPS requirements.**

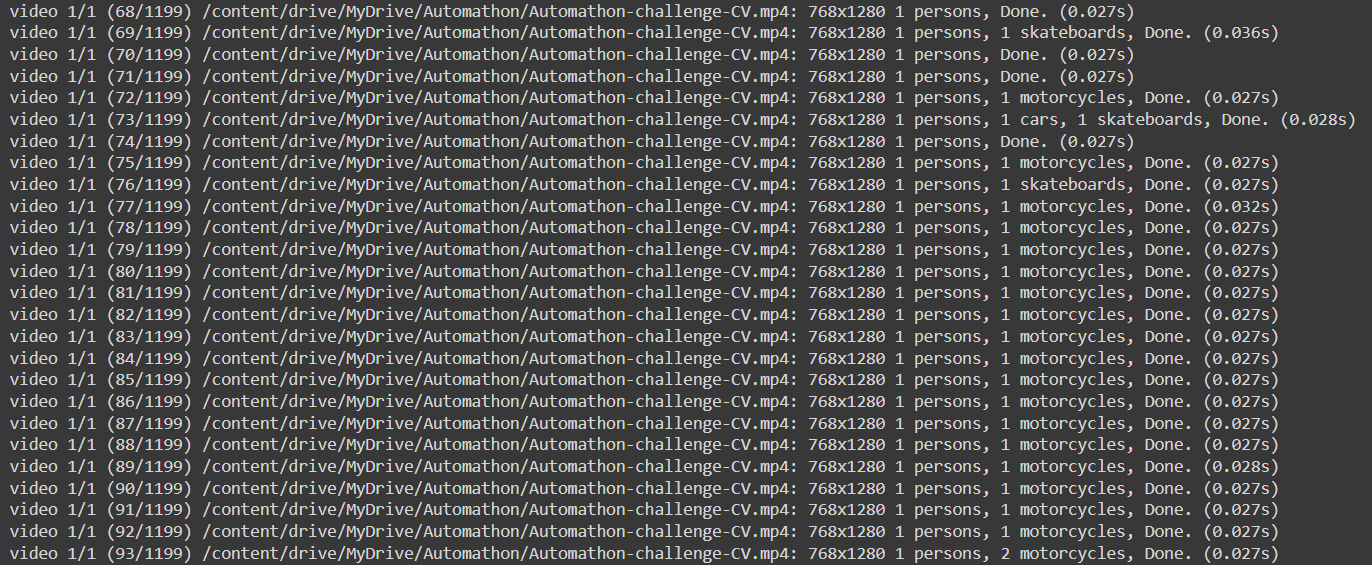
# Implementation of YOLOR

I followed the official documentation to apply the algorithm in Google Colab provided here[[9]](https://github.com/WongKinYiu/yolor). For such a powerful algorithm, their documentation is really barebones and even has a few basic code errors for Colab implementation which I will send to the creator after this project.

I initially planned to train the model from scratch on only cars and bikes so I tried downloading the COCO dataset among others but that alone was too big to fit into Google Drive’s storage systems. So I tried to do it on Pycharm but that needed me to download then parse some 50+ GB of data for 100+ epochs while simultaneously tracking it on WnB and maybe even some hyperparameter adjustments to reach the same level of accuracy as the original.

So naturally, I took the pre-trained weights and biases from the COCO dataset. Note that the number of pictures in the first 18 entries(these are the ones relevant to car automation) in this list[[10]](https://tech.amikelive.com/node-718/what-object-categories-labels-are-in-coco-dataset/#:~:text=After%20the%20observation%2C%20we%20will%20have%20the%20following%20tables%20that%20contain%20the%20comparison%20of%20object%20category%20list%20between%20the%20original%20paper%20and%20the%20dataset%20release.) combined beat a lot of other datasets specific to this domain. Even though the problem statement requires us to only detect cars and bikes, I’m leaving this in alongside it (this might increase the computation time but we know that YOLO has enough optimization to handle it).

I wanted to check the limits of this system so the first version didn’t resize the video frames (768x1280) and kept the required confidence value at 0. And the results were very shocking!



The first 2 frames took 0.051s and 0.048s to process but once stable it operated at approximately 0.027s. On frames with multiple objects appearing on screen (like the part with 3 cars) it peaked at 0.038s for one frame before returning to 0.027s.

The average computation speed is about 0.027s-0.028s but let’s take 0.030s – 0.040s for heavier environments. This translates to about **25-33 FPS** **without optimization with IoT=0.65** on a **NVIDIA K80** (free GPU provided by Google Colab) **in heavy environments.**

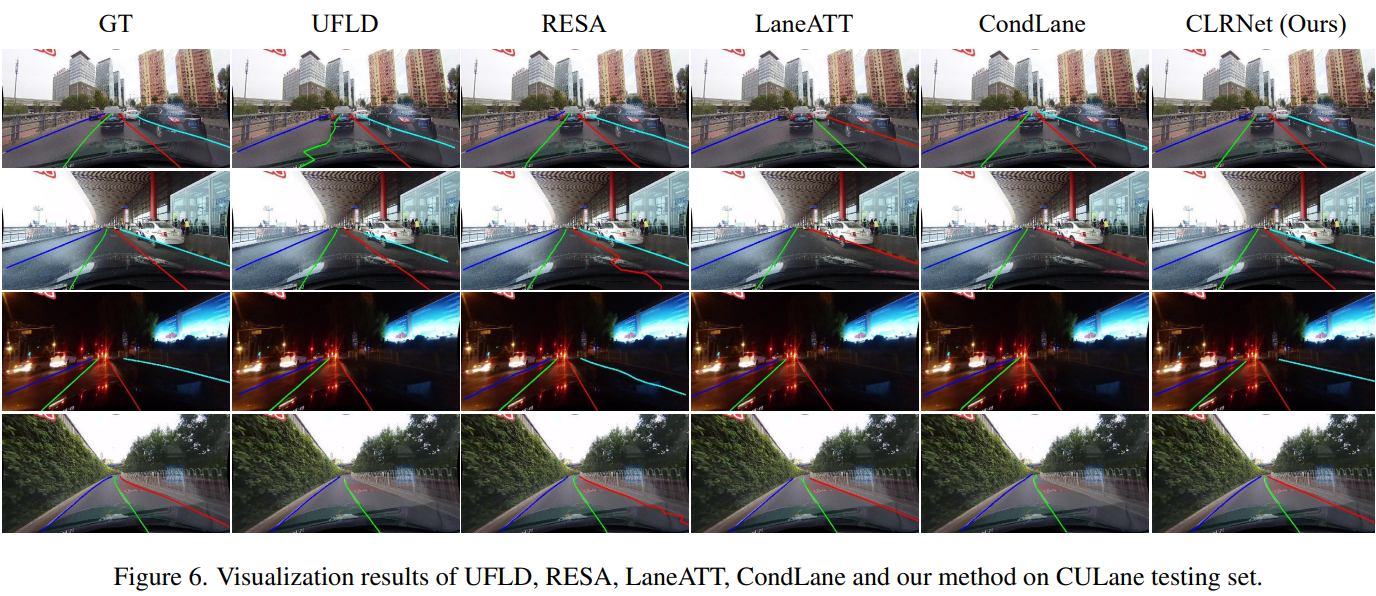
This video was processed at nearly **35-40 FPS without optimizations.**

**It is worth noting that in any attempt to detect images with viewpoint variation, deformation or occlusion the system detects a lot of false positives like fire hydrants and traffic lights.** I could just remove those from the dataset or modify their hyperparameters but they aren’t affecting the main execution of the program because they won’t be considered during decision making and we have computation power to spare.

Another reason for leaving them there is that for close range I’d rather have the algorithm detect that the object is there and get the classification wrong rather than ignoring the object entirely and ramming into it. This can easily be fixed using IR, ultrasound or LiDAR sensors but the problem statement requires us to only use this camera inside the car.

# Selection of Lane Detection Algorithm

This selection was fairly simple considering how great this paper is [[11]](https://arxiv.org/pdf/2203.10350v1.pdf). **CLRNet** dominates CULane, Tusimple and LLAMAS datasets by being the highest ranked in all 3 for the papers for which code and implementation is provided.



BUT,

This algorithm is extremely new and nobody has been able to make a good tutorial of it. I tried for several hours to be one of the first to implement it but it just wasn’t working. Perhaps another big barrier was my lack of experience in conda environments. However this repo is constantly being updated with the most recent changes being only a few days ago so we might use this eventually.

I found this paper as well [[12]](https://arxiv.org/pdf/2105.05003v2.pdf) with probably the best documentation out of all of these. I tried to simulate a Conda environment in Google Colab[attachment] which didn’t go too well. It broke my Conda environment in Windows and even Ubuntu [Video to summarize my suffering: [Video](https://drive.google.com/file/d/1CSx7gGnqNPfppnI1xJYoYfrUyEDZe1TC/view?usp=sharing)]

I have tried implementing a lot more algorithms and learnt a lot along the way. In one of them I even reached the final training statement but Google Colab just wasn’t detecting the files. I tried going through some Stackoverflow questions about it and even posted my own question but no one was able to solve it.

My laptop wasn’t making implementation either so I decided to go the traditional route by properly coding everything in OpenCV.

This is heavily inspired by the [Python plays GTA 5](https://pythonprogramming.net/game-frames-open-cv-python-plays-gta-v/) series and some other old tutorials I used to watch and what had inspired me to go through OpenCV’s documentation. Other than that, I was aware that my model would have very different values and implementations compared to this tutorial so I proceeded accordingly.

I knew that OpenCV would struggle a lot to find the correct lines in that video with the reflections and shadows but overall, it’s better than what I expected from it.

Got the same error while implementing Object Tracking. I tried making a fork from the original Github and even moving a few files around to make them more accessible but it wasn’t working. So I watched more tutorials on how to build modules but even after trying those Google Colab just wasn’t able to locate my files. Apparently, a few of my friends were facing this error too so I’ll study more about the working of Colab itself.