

# Object Detection for Autonomous Driving using YOLO algorithm.

Abhishek Sarda  
Department of ECE, ASET  
Amity University  
Noida, India

Abhisheksarda0113@gmail.com

Shubhra Dixit  
Department of ECE, ASET  
Amity University  
Noida, India  
sdixit@amity.edu

Anupama Bhan  
Department of ECE, ASET  
Amity University  
Noida, India  
abhan@amity.edu

**Abstract** - In this paper, we have presented the work that we have done pertaining to object detection for the purpose of aiding autonomous vehicles during navigation. The detections are carried out using the state-of-the-art YOLO model which is trained on a custom-made model on a custom-made dataset. The main objective of this paper is to highlight the building of the model on low-cost computing resources. After training, we have implemented the model on video data as well as on web camera.

**Keywords** - YOLO, custom training, computer vision Autonomous vehicles.

## I. INTRODUCTION

The rise and rise of research in the field of autonomous driving has led to many different techniques of trying to visualize the vicinity of the car. Building a perception around the surrounding of the subject along with comprehensive understanding of it is imperative when it comes to autonomous driving. This is one of the most prominent applications of computer vision in autonomous driving, which entails tasks like classification, localization, detection. In this paper we are only going to deal with the object detection part which in itself is a huge sub task to complete as the car needs to know the obstacles that are present in the route where its navigation has to take place. There are many object detectors that are out there which have competent results but shortcomings also exist in almost all the prominent detectors. The major problems that need to be taken care of are:

- Accuracy needs to be of paramount importance as compromising on this parameter can be very fatal for the involved stakeholders.
- Detection speed also needs to be given its due diligence as we are going to run the algorithm in real time, so it is very important to have a high frames per second detection.
- Computational resource requirement can also be very steep when it comes to accurate object detection.
- Speed of detection also needs to be given utmost importance as we are going to implement the model on video data, the detections should be simultaneous

and in synchronization with the video it is running on.



Fig1: Web Camera YOLO Object Detection

## II. LITERATURE REVIEW

### A. OverFeat

A CNN model that simultaneously performs the task of localization, detection and recognition. Amongst the most successful models used for the purpose of detection OverFeat has successfully won the 2013 ImageNet localization challenge. With a depth of eight layers along with a dependability on a scheme that overlaps boxes with detection on scales many times and iteratively generates detections based on average.

### B. VGG-16

On an attempt to take on OverFeat in the realm of object detection and classification a layered model in the form of 16 layers and 19 layers to be known as VGG 16 and VGG 19 were devised respectively. The models did exceedingly well with the extreme depth in layers that were present at their disposal and were successful in winning the IMAGENET challenge in the year 2014 and this came in as a groundbreaking success in the field of computer vision.

### C. Fast-RCNN

Improvement in the speed and detections being made by attempting to comprehend the accuracies of models that would span deeper. The method that Fast-RCNN used was by predicting through region proposals and shared its computations for the purpose of expediting the-process-of training and the model's prediction time.

#### D. YOLO (You Only Look Once)

The YOLO model does not have the highest of accuracies but has been one of the most ground-breaking revelations in computer vision as the speed of detection that it has is just astounding. The YOLO algorithm has been trained on a total of 80 different classes which basically requires a huge amount of data and extremely high computational resources and the model also runs to a considerable depth of 12 layers.

#### E. Custom Trained YOLO

In order to reduce the computational resource requirement of the YOLO model and to be able to use it for the applications that we care about we have trained the model on images of the following classes- Car, Person, Truck, Bus, Trees, Traffic Light, Traffic Sign, Bicycle. Since we decided to train an object detector for the purpose of autonomous driving we trained the model on the above classes and the implemented the model on a video footage drawn from the street and also on the Web Camera to give us the best possible intuition of the real world simulation of the model.

### III. OBJECT DETECTION

When we do object detection using a convolutional neural network, we use something called a sliding window approach. In this project we have trained the model to detect the objects that are possibly going to be found on the road by taking cropped up images of the classes of objects that we care about. So the window slides over the entire image with a small stride and returns a label of 0 or 1 for any objects that it can detect in the set of objects that it has been trained for. The size of the window keeps varying and runs multiple times through the image. But there is a huge disadvantage of this method which is the high computational cost that will be incurred because of running through each image multiple times on different size. Using a single kind of window size can result in the entire information not being grasped perfectly, so we cannot increase the strides and size of window to get through each image faster.

In the sliding window approach the window needs to go around running convolution operations in the entire image which might not be the most efficient thing to do as most parts of the image would not contain the objects that we care about. So to combat this loophole an algorithm called R-CNN was developed which would basically run the convolution operations on the regions where an object is present and therefore significantly reduces the computational efficiency of the model at hand. The way they perform the region proposals is by running a segmentation algorithm to figure out what could be objects in a given image. So when the segmentation algorithm finds different blobs in the images it then runs the classifier in it and tries to circumvent it with a bounding box thereby stating the class of the object. The only

caveat towards sticking to this approach is that this process tends to be too slow where you propose regions in an image first and run the window one by one to all the proposed regions to classify the objects.

To improve on this regard the fast R-CNN was developed which basically proposed regions in the first step and then ran convolutions simultaneously on all the regions of interest which expedited the process significantly. After that the faster R-CNN came into the scheme of things where instead of using the traditional segmentation algorithms to figure out the regions of interest where replaced by the convolutional method of proposing all regions simultaneously and classifying all regions simultaneously after that by running convolutions once again and made the process even faster but it was still a two stage process which meant that there still was a possibility of improvement and that is exactly where the single stage detector YOLO algorithm comes into the picture which is leaps and bounds ahead in the respect of speed of detecting objects.

### IV. WORKING OF YOLO

YOLO has a speed that is significantly greater than any other algorithm. To train a YOLO algorithm the first thing that we do is prepare a training set of all the classes of objects that we care about. We do not need to train the model for a background class. When the train images are passed to our model for the creation of a custom trained detector the entire process convenes in a single stage unlike the multi-stage detectors. The entire image gets segregated into equally sized grid cells and the algorithm runs simultaneously on all the cells combined. We can vary the number of anchor boxes that are formed at a given instant, the value that we have considered is 2 which essentially means that there can be 2 anchor boxes on any grid cell as illustrated in the first image below but this makes it a bit too cluttered as there are too many boxes that can be seen everywhere, to deal with this issue we apply a non-max suppression which removes all the bounding boxes with low probability values. It generates the final output for all the classes that the model has been trained for independently. As it is depicted in the images down below the first picture represents the image with all its bounding boxes and the second image represents the image after running the non-max suppression.



Fig2: Anchor Boxes and Non-Max Suppression

The encoding of an object represents the elements highlighted in the figure below where  $P_c$  represents the existence of the object followed by the coordinates of the bounding boxes

after which all the different classes that the model has been trained for get listed. Suppose that we have a  $19 \times 19$  grid along with 2 anchor boxes, the total number of bounding boxes drawn would be  $19 \times 19 \times 2 \times 13$  (last parameter is 5 + number of classes) but since there is only one object in the entire image the value of  $P_c$  in most part of the image would be 0 and rest of the parameters are going to be don't cares. Since there are two anchor boxes for each grid cell, we will also have to see which box circumvents the center of the image by checking the value of intersection over union as one of the boxes can be horizontally bent where as the other box could have a slightly different orientation. The figure down below basically shows us the encoding of each of the parameters and also gives an inkling on how the detected output should look like.

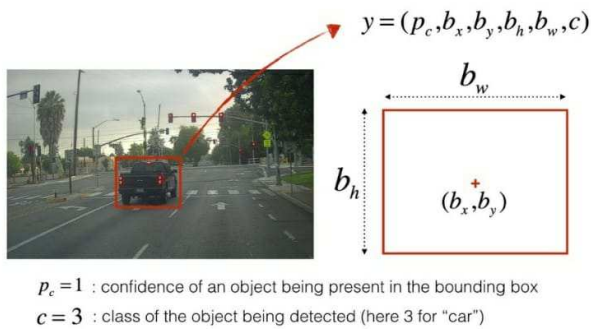


Fig4: Encoding an object

The figure down below will show the working of all the steps that are involved in the form of a flowchart. The camera captures the video from which the images get extracted and then scaled to  $416 \times 416$ . Convolutions are passed on the image frames and outputs are filtered through non max suppressions, detections are drawn on the results and the processed video file containing all the detections is obtained.

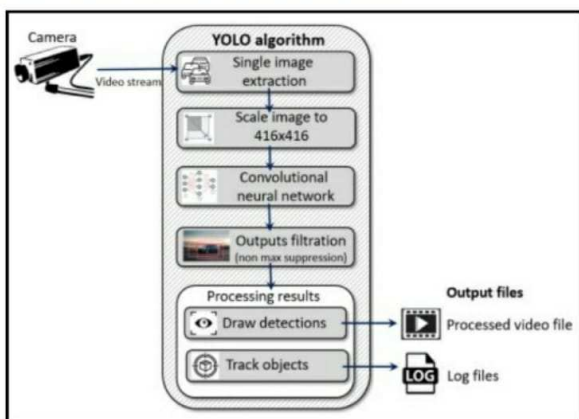


Fig3: YOLO algorithm flowchart

## V. METHODOLOGY

- Set up the OIDv4 toolkit in order to prepare the dataset.

- Download images of classes- Car, Person, Truck, Bus, Trees, Traffic Light, Traffic Sign, and Bicycle along with labels.
- Generate YOLO annotations.
- Set up required directories on Google drive after upload the zipped file of images and annotations.
- Prepare the code to train the model on Google colab.
- Store all the weights in a backup folder.
- Download the weights on local machine.
- Setup dark net, CMAKE, MS Visual Studio, open cv on local machine.
- Run the model on video data and on web camera.
- Store the results of detections on the local machine.

## VI. DATA SET

The dataset has been extracted from the Google open images dataset which contains objects of 600 categories. Since we want to work on the project of autonomous driving, we wish to train the model only on the objects that we presume could be found on the road. So the classes that we decided to choose are as follows- car, person, truck, bus, trees, traffic light, traffic sign, bicycle.

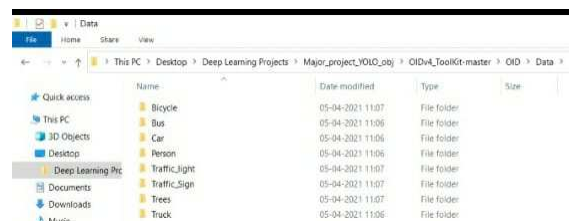


Fig5: Data Directories

## VII. CUSTOM TRAINING

To begin the custom training we start off with setting up the environment for procuring the data that the model is going to be trained on. After downloading the OIDv4 tool kit, we procure the images for the classes that we care about. The data is procured along with its corresponding labels, so that needs to be converted into YOLO understandable language which are termed as annotations. This contains the encoding of the respective classes along with the coordinate of the object on each of the images in a distinct manner. The next

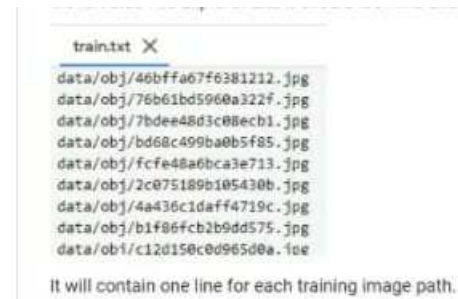
step was to compress the entire data and then upload it to the drive which is linked to the cloud notebook where the model is going to be trained.

Separate nested folders are created on the drive depending on the task that they fulfill. We create a folder that contains the names of all the classes that the model has been trained for. We write a python script to generate test.py and train.py. We then create a backup folder which keeps storing all the weights as the model gets trained. Sometimes while the training is going on there is a possibility of the kernel being stopped for some reason and that would result in the loss of all the trained weights up till that point. So we keep saving the weights in separate folders after the completion of every 1000<sup>th</sup> iteration. So if the training kernel gets interrupted for some reason or we get kicked out of the kernel because of exhausting the time that was provided to us, we can immediately restart training from the exact point that it stopped as we have all the weights saved up until that point.

After the completion of creating all the directory structure we link the colab notebook where we are supposed to train the model with our drive. We started by cloning the dark net repository in which the YOLO algorithm has been written, we set up the entire environment for dark net thereby installing all the required libraries. The next step is to test whether the algorithm is functioning and for that we download the YOLOv3 weights and run it on a test image after the installation of Open CV.

The next step is to begin the training of the model which initially requires us to copy the zipped file on the virtual machine and unzip it while the kernel is running. Now we move towards setting all the required hyper parameters to the optimal values. For the hyper parameter tuning we prefer the value of subdivisions to be 16 although this needs to be tried on a hit and trail method. Scale down the size of all the images to 416\*416 and the batch size to be 64. The maximum batches value has been chosen based on the formula no. of classes times 2000, so in this case it turns out to be 16000. We have chosen different kinds of activation functions throughout like leaky Relu, linear, etc.

Now we move towards generating the text files for all the images by writing the python script. We represent each of the images in our training set respectively.



```
train.txt X
data/obj/46bffa67f6381212.jpg
data/obj/76b61bd5960a322f.jpg
data/obj/7bdee48d3c08ecb1.jpg
data/obj/bd68c499ba0b5f85.jpg
data/obj/fcfe48a6bca3e713.jpg
data/obj/2c075189b105430b.jpg
data/obj/4a436c1daff4719c.jpg
data/obj/b1f06fcb2b9dd575.jpg
data/obj/c12d150c0d965d0a.jpg
It will contain one line for each training image path.
```

Fig6: Train.txt

Now we start the training of the model and let it train for 6000 iterations. After the training is completed we produce the graph of iterations with respect to the images.

To complete the main objective of the project we need to run the trained weights model on real world video frames data and also on web camera.

We start of by downloading NVIDIA GPU drivers along with CMAKE and then install OPENCV in our local machine. After setting up the entire dark net environment on our local machine, we download the custom trained weights from drive and then convert the weights into tensor flow understandable form. Here are the screen shots of the output on video frames data and on video run on Web camera on the custom trained YOLO object detection model. The first image shows the screen shot of a video taken from a busy street running the model and the second image shows the custom model being run on web camera.

## VIII. RESULT

The video data is streaming at 4.06 frames per second whereas the web camera is running at 3.31 frames per second as it has been depicted down below.



Fig7: Road Video Screen Shot



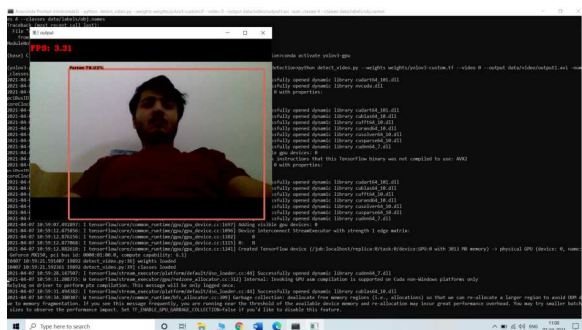


Fig8: Web Camera Video frame Screen Shot

This graph basically shows us the final result that was procured after the model training was completed. The graph is plotted between the loss function and no. of iterations. The loss stops to decrease after the 3000<sup>th</sup> iteration, so we halt the training there by bringing in early stopping so that the model does not perform over fitting.

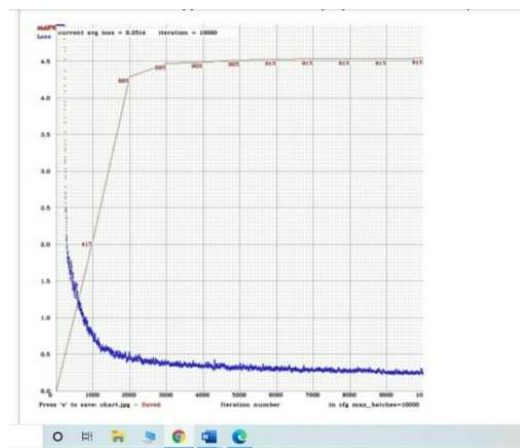


Fig9: Loss Vs Iterations

The web camera detects the frame at 78.21% and a detection speed of 3.31 FPS.

The mAP value after running the model on the above classes was found close to 74%.

## IX. CONCLUSION

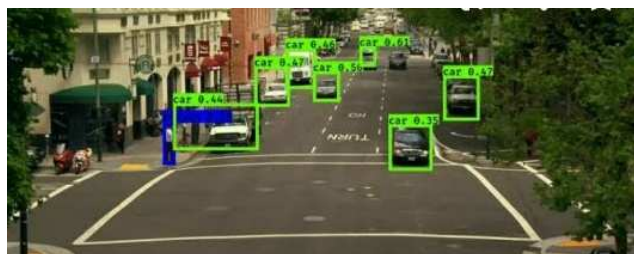


Fig10: Cars Detected through Custom Detection

The custom trained YOLO model has been trained on the above mentioned classes Car, Person, Truck, Bus, Trees, Traffic Light, Traffic Sign, and Bicycle. And successfully

implemented on video frames on the local machine. The improvements that we realise could be made here would be to improve on the frames per second value and better the accuracy.

The image data is detected at a speed of 93 milli second.

## REFERENCES

- [1] Felzenszwalb, P. F., Girshick, R. B., McAllester, D., & Ramanan, D. (2010). Object detection with discriminatively trained part-based models. *IEEE transactions on pattern analysis and machine intelligence*, 32(9), 1627–1645.
- [2] Goodfellow, I., Bengio, Y., Courville, A., & Bengio, Y. (2016). *Deep learning* (Vol. 1). Cambridge: MIT press.
- [3] Girshick, R. (2015). Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision* (pp. 1440–1448).
- [4] Ren, S., He, K., Girshick, R., & Sun, J. (2015). Faster rcnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems* (pp. 91–99).
- [5] Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 779–788).
- [6] Redmon, J., & Farhadi, A. (2018). Yolo v3: An incremental improvement. *arXiv preprint arXiv:1804.02767*.
- [7] Springenberg, J. T., Dosovitskiy, A., Brox, T., & Riedmiller, M. (2014). Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*.
- [8] X. Chen, H. Ma, J. Wan, B. Li, and T. Xia, “Multi-view 3d object detection network for autonomous driving,” in *CVPR*, 2017.
- [9] A. Dundar, J. Jin, B. Martini, and E. Culurciello, “Embedded streaming deep neural networks accelerator with applications,” *IEEE Trans. Neural Netw. & Learning Syst.*, vol. 28, no. 7, pp. 1572–1583, 2017.
- [10] R. J. Cintra, S. Duffner, C. Garcia, and A. Leite, “Low-complexity approximate convolutional neural networks,” *IEEE Trans. Neural Netw. & Learning Syst.*, vol. PP, no. 99, pp. 1–12, 2018.
- [11] Jiale Cao, Yanwei Pang, Jungong Han, and Xuelong Li. Hierarchical shot detector. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pages 9705–9714, 2019. 12
- [12] Ping Chao, Chao-Yang Kao, Yu-Shan Ruan, ChienHsiang Huang, and Youn-Long Lin. HardNet: A low memory traffic network. *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2019. 13
- [13] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L Yuille. DeepLab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected CRFs. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 40(4):834–848, 2017. 2, 4
- [14] Pengguang Chen. GridMask data augmentation. *arXiv preprint arXiv:2001.04086*, 2020. 3
- [15] Yukang Chen, Tong Yang, Xiangyu Zhang, Gaofeng Meng, Xinyu Xiao, and Jian Sun. DetNAS: Backbone search for object detection. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 6638–6648, 2019. 2
- [16] Jiwoong Choi, Dayoung Chun, Hyun Kim, and HyukJae Lee. Gaussian YOLOv3: An accurate and fast object detector using localization uncertainty for autonomous driving. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pages 502–511, 2019. 7
- [17] Jifeng Dai, Yi Li, Kaiming He, and Jian Sun. R-FCN: Object detection via region-based fully convolutional networks. In *Advances in Neural Information Processing Systems (NIPS)*, pages 379–387, 2016. 2
- [18] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A large-scale hierarchical image database. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 248–255, 2009. 5