CAREER EPISODE 1

AUTOMATIC LICENSE PLATE RECOGNITION

INTRODUCTION

CE 1.1

After graduating from college, I was accepted to [redacted] Computer Science and Engineering program. One of the projects I worked on for this degree was developing an automatic license plate recognition, which I was asked to work on in my last two semesters, under the course title "Final Year Project I & II". After discussing the idea with one of my lecturers, he offered a lot of suggestions to improve the idea. I read all the literature that was available to me at the university's library and labs.

BACKGROUND

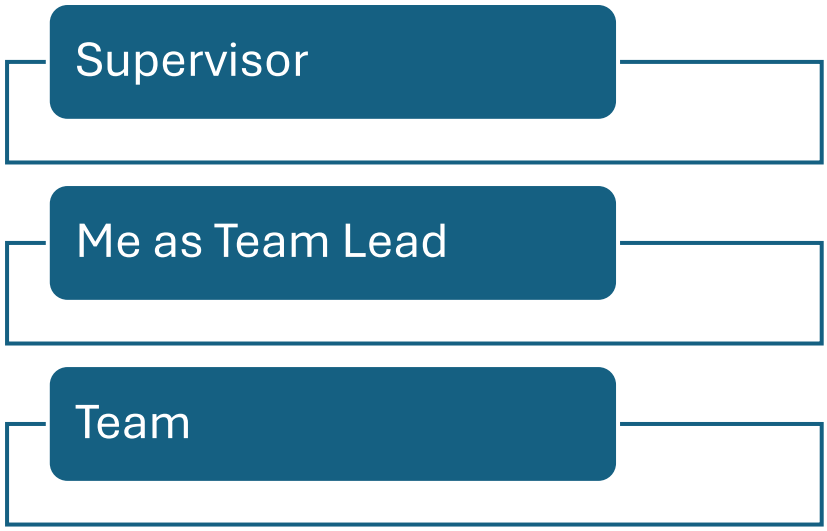
CE 1.2.1

Identifying the license plate from the entire image and subsequently identifying its characters was the goal of this project. The goal of numerous internet platforms and Github open-source code was to identify and detect license plates from international datasets, such as automobiles. Unfortunately, none of the online tools that were specifically designed to detect and recognize the license plate of indian automobiles were available. In order to effectively perform ALPR on [redacted] datasets, I sought to develop a framework that was exclusively restricted to [redacted] automobiles.

CE 1.2.2

After approving my study proposal, my supervisor recommended that I do a more complete review of the literature. I sought help from my course lecturer and supervisor to create a project proposal. I reviewed relevant senior batch project reports and met with my mentor to draft a proposal letter. The project was approved by the department head, and my mentor encouraged me to take charge of the team, enhancing their design.

PROJECT REPORTHING HIERARCHY



PERSONAL ENGINEERING ACTIVITY

CE 1.3.1

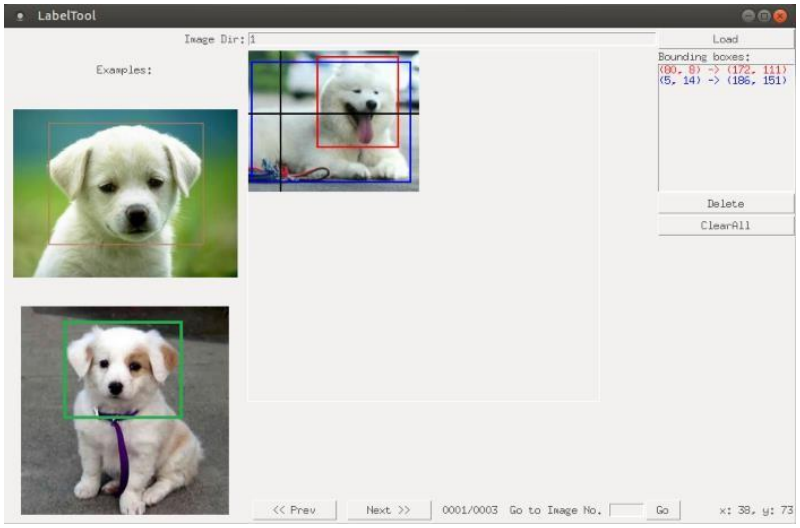
Before starting to work on this automatic number-plate recognition, I decided to do a thorough literature review. I wanted to develop a system that could detect license plates from images. YOLO was a neural network that can detect objects and locations in one pass, dividing images into multiple S*S grids. I predicted bounding boxes and suppresses low-scoring ones using Nonmax suppression. I opted to use NVIDIA CUDA that had a GPU-accelerated library for deep neural networks, offering built-in implementations for routines like pooling, forward and backward convolution, activation, and normalization. I decided to use google colab which was a free cloud service and it has been helping engineers with Python programming and building deep learning applications using popular libraries like Tensorflow, Pytorch, Keras, and OpenCV.

CE 1.3.2

I came to know that it was unique in providing completely free GPU, but it has limitations on usage. Once the GPU was exhausted, it displayed a limit usage error. I have utilized free GPU service and the Darknet framework for YOLO to train a custom dataset for recognizing License plates. My first task was to prepare dataset. The YOLOv3 dataset for training a detector consisted of two components: images and labels. Each image was linked to a.txt label file, providing object class, coordinates, width, and height information as shown:

<object-class> <x_center> <y_center> <width> <height>

To generate label files from images, I used GUI software like Yolo_label, OpenLabeling, Yolo_mark, and BBox-Label-Tool to mark bounding boxes around the object, resulting in label files corresponding to the images.

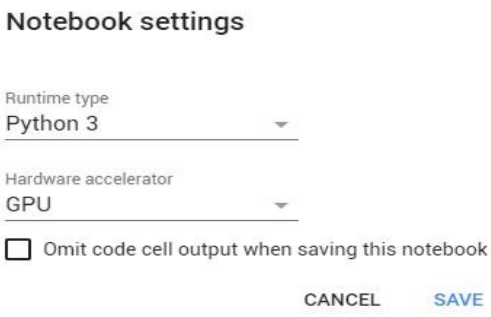


CE 1.3.3

I needed car images and license plate images. I started annotating car image and text described manual labeling process for only one class of License Plate from an image using labelling software like Yolo_label. I generated a text file with license plate coordinates, creating a bounding box during labelling, which may vary based on the number of plates in a single image. For annotating license plate images, I used text outline method for labelling, where each character bounds a box, with the number of bounding boxes determined by the number of License Plate characters. To recognize characters from the License Plate, I needed to annotate for 36 classes, including 26 letters (A-Z) and 10 numbers (0-9).

CE 1.3.4

I knew that labelling system uses numbers 0-25 for alphabets A-Z and 26-35 for numbers 0-9. The convention accommodated alphabets A-Z and numbers 0-9. To start working in Google Colab, I mounted Google Drive in my Colab environment by clicking a button. The system was used for character recognition of the License plate. My main goal was to create a comprehensive labelling system that accurately represented each character. To switch from CPU to GPU, I navigated to change runtime type and select GPU as the hardware accelerator.



I developed YOLOv3 using Darknet, which was an open-source neural network framework for detector training. I downloaded Darknet by cloning the Github repo of pjreddie in Google Colab by using the following command:

```
!git clone https://github.com/pjreddie/darknet.git
```

CE 1.3.5

Next, I downloaded pre-trained model on Imagenet directly and started uploading custom dataset images and labels. I took google colab environment inside darknet folder. To enable the program to access the images and labels, I needed it to move them into the darknet/data folder, which can only be done when both are in the data folder within the darknet folder. After setting up Darknet and uploading a custom dataset, I converted the darknet file into an executable to inform the framework about available hardware and training steps. To do this, I made changes to the Makefile in the darknet folder. As Google Colab was a free GPU service, I made the changes to the Makefile i.e. set GPU=1, set CUDNN=1, set OPENCV=1.

```
GPU=1
CUDNN=1
OPENCV=1
OPENMP=0
DEBUG=0

ARCH= -gencode arch=compute_30,code=sm_30 \
      -gencode arch=compute_35,code=sm_35 \
      -gencode arch=compute_50,code=[sm_50,compute_50]
      -gencode arch=compute_52,code=[sm_52,compute_52]
#      -gencode arch=compute_20,code=[sm_20,sm_21] \ T

```

CE 1.3.6

I knew from my previous knowledge that GPU, CUDNN and OPENCV were all initially set to 0 and I changed them to 1. Now to convert this darknet directory into an executable I run the make command in Google Colab. The make command created a darknet executable within the darknet folder, along with directories of obj containing information about the Neural Network's basic architecture, such as activation kernel, activation layer, max pooling, convolutional layer, yolo layer, and fully connected layer. Additionally, a directory of backup folder was created, providing checkpoints for saving results after iterations.

```
!ls
```

```
backup   examples       LICENSE       LICENSE.meta  obj        scripts
cfg      include        LICENSE.fuck  LICENSE.mit   python     src
darknet  libdarknet.a   LICENSE.gen   LICENSE.v1    README.md
data     libdarknet.so  LICENSE.gpl   Makefile      results
```

CE 1.3.7

I knew that I needed an obj.name file for detecting license plate using single class. It stored the name of an object to be trained for detection.

```
with open('data/obj.names', 'w') as out:
  out.write('Licenseplate')
```

I used this to recognizeable characters and it contained 36 different classes for character recognition, including 26 alphabets and 10 numbers. To generate this file, I used the following code:

```
with open('data/obj.names', 'w') as out:
  out.write('A\n')
  out.write('B\n')
  out.write('C\n')
  out.write('D\n')
  out.write('E\n')
  out.write('F\n')
  out.write('G\n')
  out.write('H\n')
  out.write('I\n')
  out.write('J\n')
  out.write('K\n')
  out.write('L\n')
  out.write('M\n')
  out.write('N\n')
  out.write('O\n')
  out.write('P\n')
  out.write('Q\n')
  out.write('R\n')
  out.write('S\n')
  out.write('T\n')
  out.write('U\n')
  out.write('V\n')
  out.write('W\n')
  out.write('X\n')
  out.write('Y\n')
  out.write('Z\n')
```

```
  out.write('0\n')
  out.write('1\n')
  out.write('2\n')
  out.write('3\n')
  out.write('4\n')
  out.write('5\n')
  out.write('6\n')
  out.write('7\n')
  out.write('8\n')
  out.write('9\n')
```

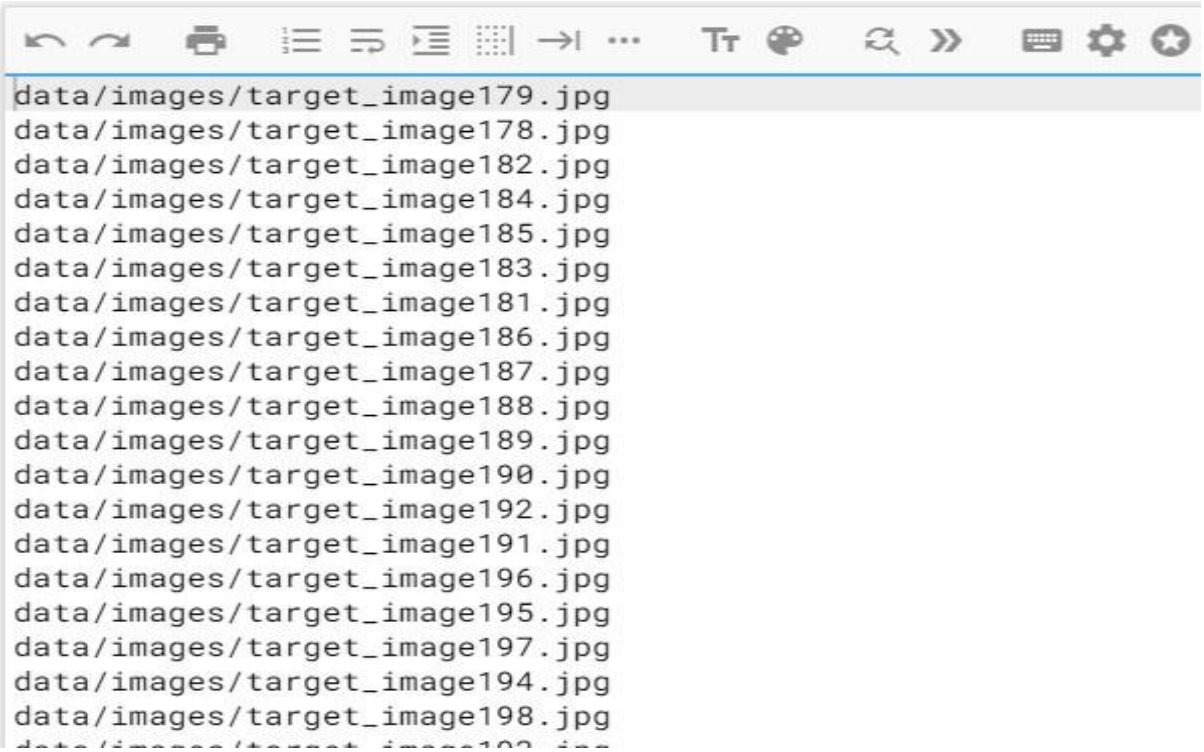
After this, I started creating a train.txt file which was used to train a classifier for license plate detection as I needed to organize the custom images of cars into a single file with the location of each image. For example, I have uploaded 3000 YOLO-format images to train for license plate detection. The train.txt file contained the path and name of each image. This was done using Google Colab's code.

```
#write train file (just the image list)
import os

with open('data/train.txt', 'w') as out:
  for img in [f for f in os.listdir('data/images') if f.endswith('jpg')]:
    out.write('data/images/' + img + '\n')
```

CE 1.3.8

Following code generated an empty train.txt file, then used a for loop to iterate over all images in the data/images folder. If the filename ended with.jpg, it updated the name and path of the image to the empty train.txt file. All images in the data/images folder were written in.jpg format, as shown when using License Plate images for character recognition


```
#write train file (just the image list)
import os

with open('data/train.txt', 'w') as out:
  for img in [f for f in os.listdir('data/images') if f.endswith('jpg')]:
    out.write('data/images/' + img + '\n')
```

```
data/images/target_image179.jpg
data/images/target_image178.jpg
data/images/target_image182.jpg
data/images/target_image184.jpg
data/images/target_image185.jpg
data/images/target_image183.jpg
data/images/target_image181.jpg
data/images/target_image186.jpg
data/images/target_image187.jpg
data/images/target_image188.jpg
data/images/target_image189.jpg
data/images/target_image190.jpg
data/images/target_image192.jpg
data/images/target_image191.jpg
data/images/target_image196.jpg
data/images/target_image195.jpg
data/images/target_image197.jpg
data/images/target_image194.jpg
data/images/target_image198.jpg
```

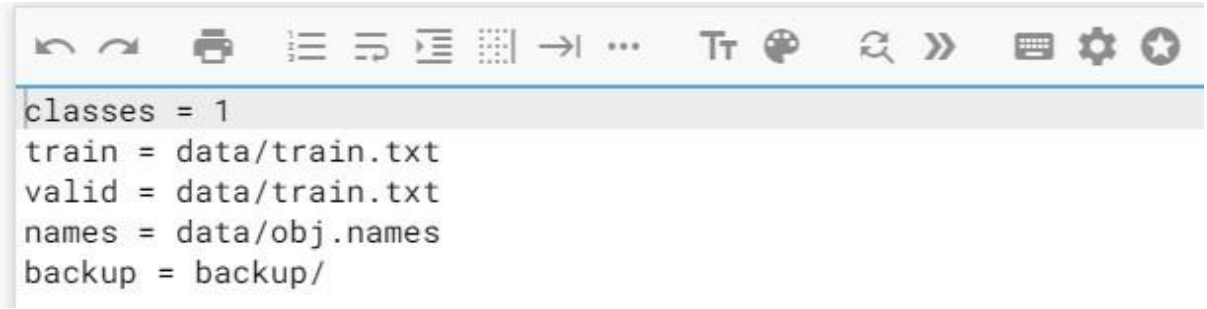
After this, I generated obj.data file, with a.data extension, and stored information about the number of classes, the path of both the training and validation dataset, the name of the classes

being trained for, and the path of the backup folder. When training was performed with user specified number of iterations, than the file was stored. In this scenario, I generated obj.data file separately for License Plate detection and Character recognition, as they were being trained separately. The code snippet outlined a method for License Plate detection from an entire image, involving only one class and specifying the paths of train.txt, obj.names, and backup folders in the darknet/data folder and the backup folder in the darknet folder.

```
with open('data/obj.data', 'w') as out:
  out.write('classes = 1\n')
  out.write('train = data/train.txt\n')
  out.write('valid = data/train.txt\n')
  out.write('names = data/obj.names\n')
  out.write('backup = backup/')
```

CE 1.3.9

I wrote the above snippet of code following .data file and generated in darknet/data folder:



```
classes = 1
train = data/train.txt
valid = data/train.txt
names = data/obj.names
backup = backup/
```

In one of my courses, I studied that in order to configure parameters and basic settings for different computer programs, user applications, server operations, and operating system settings, I needed to look into the significance of configuration files. The YOLOv3 framework, originally trained on 80 classes, I wanted to adjust yolov3 config file parameters to be compatible with my custom dataset. This was done for License Plate detection and Character Recognition, ensuring compatibility with my custom dataset. In this config file, I defined batch size, subdivisions, momentum and decay, classes, activation, max_batches, steps, and filters. I knew that batch size was dependent on GPU memory and was determined through trial and error. Subdivisions were used for parallel processing.

CE 1.3.10

Momentum and decay were learning parameters which were initially 0.9 and 0.0005 respectively. Classes were specified in the YOLO layer of the CNN.

```
batch=16
subdivisions=4
```

Similarly, I set image width and height to 416:

```
width=416
height=416
```

Next, I set max_batches for License plate detection as follows:

```
max_batches = 9000
policy=steps
steps=7200,8100
```

I adjusted number of classes to 1 i.e. in YOLO layer. The standard for computing number of filters as described in pjreddie github repo was as follows

$$No.of\ filters = (No.of\ classes + 5) * 3 = (1 + 5) * 3 = 18$$

For, character recognition, I set batches with 36 classes and number of filters were determined using following formula:

$$No.of\ filters = (No.of\ classes + 5) * 3 = (36 + 5) * 3 = 123$$

CE 1.3.11

My supervisor suggested me to look into epoch which was a machine learning term indicating the number of passes the algorithm has made over a dataset, typically grouped into batches. These passes are calculated using parameters set in the config file. To calculate the number of epochs in darknet following formula was applied:

$$No.of\ epochs = (max\_batches * batch)\ /\ (no.of\ training\ images)$$

For License Plate detection, I set parameters for calculating the number of epochs i.e. max_batches=9000, batch=16 & training images=3000 and calculations are shown below:

$$No.of\ epochs = (9000 * 16)/3000 = 48\ epochs$$

For License plate character recognition, I have set parameters through which I calculated the number of epochs i.e. max_batches=6000, batch=16 & training images=1179.

$$No.of\ epochs = (6000 * 16)/1179 = 81\ epochs\ (approx)$$

CE 1.3.12

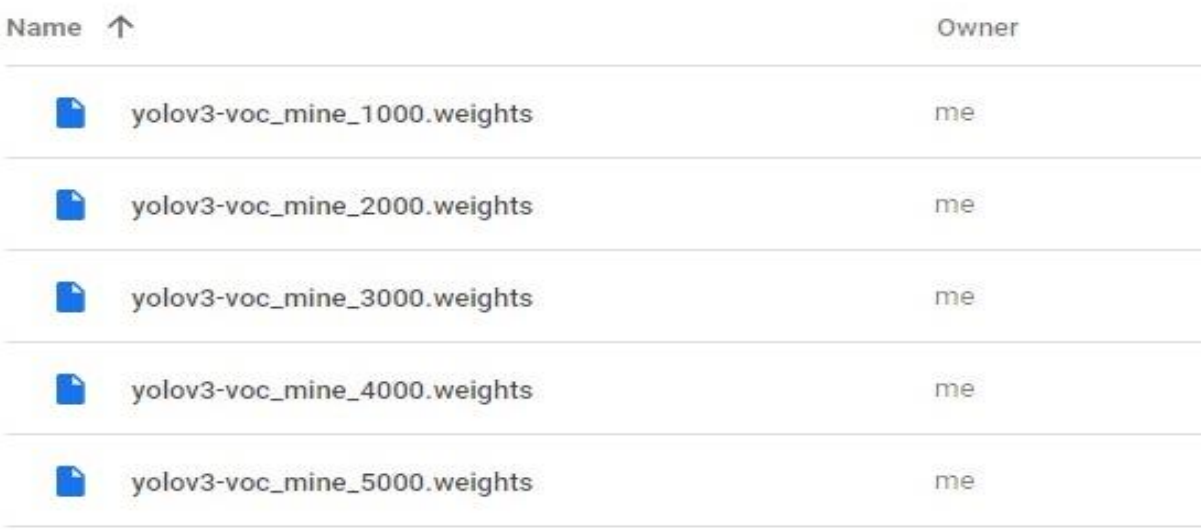
I saved the model's weight every 1000 iterations as COLAB only allowed 12 hours of training with the browser open, allowing the saved weight to be reloaded later. After this, I started training my model as shown below:

```
!./darknet detector train data/obj.data cfg/yolov3-voc_mine.cfg darknet53.conv.74
```

The training started after the following command executes. The output of this cell is shown as follows:

```
!./darknet detector train data/obj.data cfg/yolov3-voc_mine.cfg darknet53.conv.74

Streaming output truncated to the last 5000 lines.
Region 94 Avg IOU: 0.845595, Class: 0.994386, Obj: 0.958330, No Obj: 0.014558, .5R: 1.000000, .75R: 0.956522,  count: 23
Region 106 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000001, .5R: -nan, .75R: -nan,  count: 0
Region 82 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000035, .5R: -nan, .75R: -nan,  count: 0
Region 94 Avg IOU: 0.870964, Class: 0.996416, Obj: 0.995301, No Obj: 0.012414, .5R: 1.000000, .75R: 1.000000,  count: 22
Region 106 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000001, .5R: -nan, .75R: -nan,  count: 0
Region 82 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000046, .5R: -nan, .75R: -nan,  count: 0
Region 94 Avg IOU: 0.845922, Class: 0.996055, Obj: 0.949339, No Obj: 0.016115, .5R: 1.000000, .75R: 0.958333,  count: 24
Region 106 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000002, .5R: -nan, .75R: -nan,  count: 0
5649: 0.306326, 0.411549 avg, 0.000010 rate, 0.956776 seconds, 90384 images
Loaded: 0.000049 seconds
Region 82 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000006, .5R: -nan, .75R: -nan,  count: 0
Region 94 Avg IOU: 0.849675, Class: 0.996041, Obj: 0.998149, No Obj: 0.011815, .5R: 1.000000, .75R: 1.000000,  count: 21
Region 106 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000001, .5R: -nan, .75R: -nan,  count: 0
Region 82 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000004, .5R: -nan, .75R: -nan,  count: 0
Region 94 Avg IOU: 0.849092, Class: 0.997561, Obj: 0.994963, No Obj: 0.015293, .5R: 1.000000, .75R: 1.000000,  count: 23
Region 106 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000001, .5R: -nan, .75R: -nan,  count: 0
Region 82 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000005, .5R: -nan, .75R: -nan,  count: 0
Region 94 Avg IOU: 0.829430, Class: 0.992694, Obj: 0.994297, No Obj: 0.013777, .5R: 1.000000, .75R: 0.916667,  count: 24
Region 106 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000002, .5R: -nan, .75R: -nan,  count: 0
Region 82 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000005, .5R: -nan, .75R: -nan,  count: 0
Region 94 Avg IOU: 0.758379, Class: 0.912337, Obj: 0.927635, No Obj: 0.012183, .5R: 1.000000, .75R: 0.545455,  count: 22
Region 106 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000001, .5R: -nan, .75R: -nan,  count: 0
5650: 0.387524, 0.409146 avg, 0.000010 rate, 0.949588 seconds, 90400 images
Resizing
320
Loaded: 0.000059 seconds
Region 82 Avg IOU: -nan, Class: -nan, Obj: -nan, No Obj: 0.000161, .5R: -nan, .75R: -nan,  count: 0
Region 94 Avg IOU: 0.824450, Class: 0.970597, Obj: 0.974729, No Obj: 0.017334, .5R: 1.000000, .75R: 0.875000,  count: 24
```

The program now stored the trained model results after 1000 iterations if the number of iterations were less than 10000, with License Plate detection having a 9000-iteration count and Character recognition having a 6000-iteration count, storing intermediate results in the darknet/backup folder.



CE 1.3.13

If training stopped due to connectivity or other issues, I was able to resume it from where it left off. For example, if I trained up to 5000 iterations, I can resume the training from the 5000th iteration and so on until I reach specified number of iterations. After executing all steps, I was ready to start training again. To do this, I used the following command:

```
!./darknet detector train data/obj.data cfg/yolov3-voc_mine.cfg yolov3-voc_mine_5000
```

The command now trained from the previous pre-trained model, darknet53.conv.74, instead of darknet53.conv.74. If successful, a final weight final named yolov3-voc_mine_final was obtained, which was a.weights file in the standard format of the darknet framework. After obtaining training weight, I used OpenCV to load YOLOv3 architecture and used this weight to deliver prediction. First of all, I create a Ananconda environment for Github repo. I then open object_detection_yolo.py file and then install all the required packages to get my Github repo up and running. Following are the packages that I need to install

```
import cv2 as cv
import argparse
import sys
import numpy as np
import os.path
from PIL import Image
```

CE 1.3.14

My next task was to initialize parameters confidence thresholds and non max suppression.

```
confThreshold = 0.5  # Confidence threshold
nmsThreshold = 0.4  # Non-maximum suppression threshold
```

During image testing, multiple bounding boxes were detected, with a confidence threshold of 0.5. To ensure valid detection, non-valid bounding boxes were suppressed using non-max suppression on all detected boxes, retaining only those with a threshold value of 0.4. I loaded the network using configuration and weight files. To ensure correct application of weights and configuration files for detection and character recognition, I changed the names of these files to identify which one belonged to which procedure.

```
def drawPred(classId, conf, left, top, right, bottom):
    # Draw a bounding box.
    #    cv.rectangle(frame, (left, top), (right, bottom), (255, 178, 50), 3)
    cv.rectangle(frame, (left, top), (right, bottom), (0, 255, 0), 3)

    label = '%.2f' % conf


    # Get the label for the class name and its confidence
    if classes:
        assert(classId < len(classes))
        label = '%s: %s' % (classes[classId], label)

    # Display the label at the top of the bounding box
    labelSize, baseLine = cv.getTextSize(
        label, cv.FONT_HERSHEY_SIMPLEX, 0.5, 1)
    top = max(top, labelSize[0])
    cv.rectangle(frame, (left, top - round(1.5*labelSize[1])), (left + round(
        1.5*labelSize[0]), top + baseLine), (255, 0, 255), cv.FILLED)
    #cv.rectangle(frame, (left, top - round(1.5*labelSize[1])), (left + round(1
    cv.putText(frame, label, (left, top),
               cv.FONT_HERSHEY_SIMPLEX, 0.70, (255, 255, 255), 2)
```

CE 1.3.15

The function predicted the output frame by drawing a bounding box over the detected data using the cv. rectangle command. It provided coordinates, color, and scale information. The class name label was displayed at the top of the bounding box. The final command specified font, scaling factor, and text color for the label. I removed the bounding boxes with low confidence using non-max suppression. I opted to scan all network bounding boxes and selected high-confidence ones. I assigned the box's class label to the highest-scoring class. After this, I placed a threshold to retain only those bounding boxes whose confidence was greater than the confidence threshold which I set earlier to be 0.5.

```
            if confidence > confThreshold:
                center_x = int(detection[0] * frameWidth)
                center_y = int(detection[1] * frameHeight)
                width = int(detection[2] * frameWidth)
                height = int(detection[3] * frameHeight)
                left = int(center_x - width / 2)
                top = int(center_y - height / 2)
                classIds.append(classId)
                confidences.append(float(confidence))
                boxes.append([left, top, width, height])
```

CE 1.3.16

I performed non-max suppression to eliminate redundant overlapping boxes with lower confidences. This is shown as follows:

```
indices = cv.dnn.NMSBoxes(boxes, confidences, confThreshold, nmsThreshold)
for i in indices:
    i = i[0]
    box = boxes[i]
    left = box[0]
    top = box[1]
    width = box[2]
    height = box[3]
    drawPred(classIds[i], confidences[i], left,
             top, left + width, top + height)
```

After completing all steps and providing parameters like weights and a config file, I can proceed to perform testing on my test images by opening the anaconda prompt for the program environment. I performed test on a single image file:

```
python object_detection_yolo.py --image=bird.jpg
```

To test on a single video file, I used the following format:

```
python object_detection_yolo.py --video=cars.mp4
```

The system processed video frames by frame, detecting each frame in 30-60 frames per second, to provide the necessary detection for modern video formats.

CE 1.3.17

To detect License plate, I run the following command:

```
(code1) C:\Users\hp\Desktop\code#1\yolo-license-plate-detection-master>python object_detection_yolo.py --image=82.jpg
```

Which showed it's predicted coordinates in an image as follows:

```
out.shape :  (507, 6)
out.shape :  (2028, 6)
0.9913214  -  0.0  - th :  0.5
[0.6481977  0.6899465  0.18706149 0.07469284 0.9913214  0.        ]
0.9066751  -  0.0  - th :  0.5
[0.6555984  0.68951863 0.21321076 0.09333874 0.9066751  0.        ]
0.995059  -  0.994779  - th :  0.5
[0.64474523 0.6943707  0.21060643 0.0835104  0.995059   0.994779  ]
0.69084656  -  0.0  - th :  0.5
[0.65586334 0.69388366 0.1721008  0.09398007 0.69084656 0.        ]
out.shape :  (8112, 6)
```

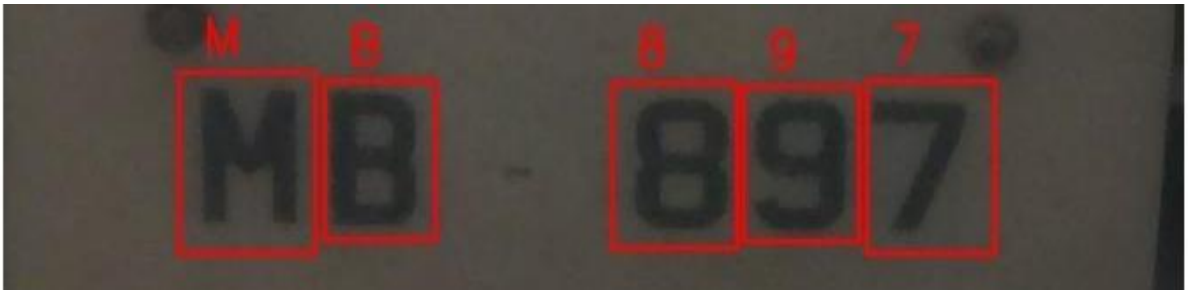
The output image was stored in the current Anaconda prompt directory, ensuring the correct weights and config file path are provided after training for License plate detection.

```
Done processing !!!
Output file is stored as  82_yolo_out_py.jpg
```

To recognize characters from the License Plate I run the following command:

```
(code1) C:\Users\hp\Desktop\code#1\yolo-license-plate-detection-master>python object_detection_yolo.py --image=target_image7.jpg
```

The command outputs coordinates and confidence scores for valid characters in License plate, requiring the path of weights and the config file for character recognition. An image output is shown as follows:



CE 1.3.18

My next task was to do quantification that converted results into a measurable form, such as comparing ground truth results with detected number plates and representing it through a precision recall curve, requiring a defined term for easy understanding. I looked into IOU

(intersection over union) to evaluate the overlapping in ground area & detected box while setting a threshold. If IOU exceeds it, the detected detection was considered true.

$$\text{IOU} = \frac{\text{area}\,(B_p \cap B_{gt})}{\text{area}\,(B_p \cup B_{gt})}$$

I considered that if Detection with IOU was greater than threshold than this was a correct detection. If Detection with IOU was lesser than threshold than this was a correct detection. To comprehend the precision recall curve, it was crucial to comprehend the distinction between precision and recall.

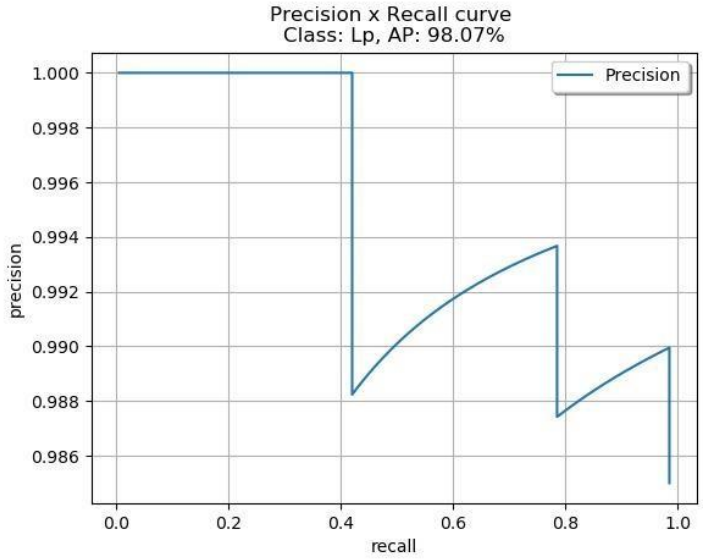
CE 1.3.19

I was familiar from one of the classes that a model's accuracy was its capacity to recognize only the necessary things. It was calculated as the proportion of accurate positive projections and was based on:

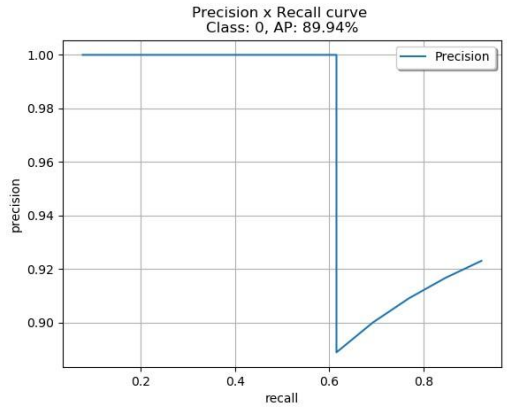
$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} = \frac{\text{TP}}{\text{all detections}}$$

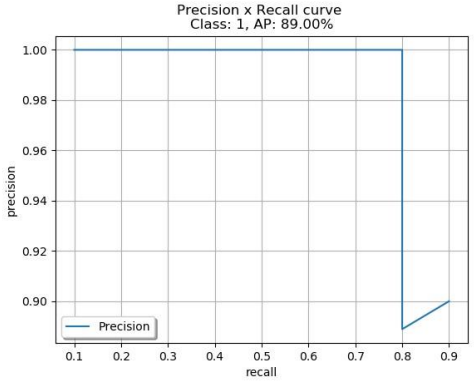
Whereas, based on the proportion of true positives found among the relevant ground truths, recall was a model's ability to find all pertinent cases.

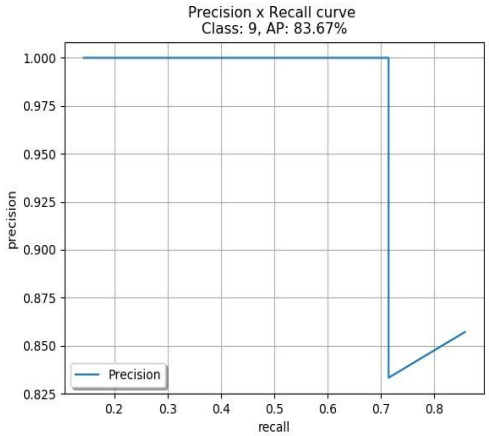
$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} = \frac{\text{TP}}{\text{all ground truths}}$$

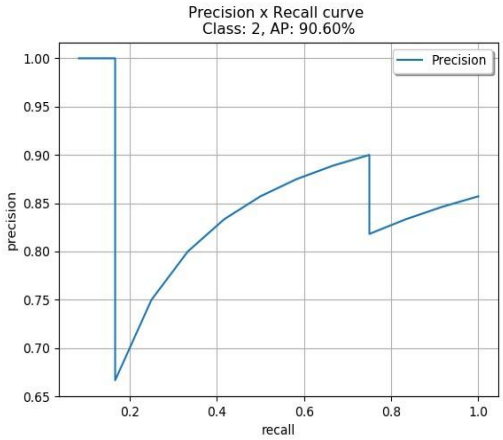


The project repeated steps for character recognition and license plate detection, obtaining precision recall for each class and a single precision recall curve for the license plate detection case.









SUMMARY

1.4

In this project, I worked on automatically identifying license plates using images. I was able to successfully identify characters and alphabets with the help of classes. I used an open-source platform which helped me in determining international and national datasets. I knew there were no such system was developed in [redacted]. I obtained precision recall for each class and generated precision recall curves.