Division of computer Engineering
School Of Engineering
Cochin University Of Science And Enginerring
Kochi - 682022

CERTIFICATE

Certified that this is a bonafide record of the mini project titled

Vehicle Entry Monitoring System

Presented by

12180063 Naveen Kamal P V
12180066 Prakhalbh Santhosh M
12180074 Sanjay K Shajil
12180081 Sidharth Sunil

of Sixth semester B.Tech in Computer Science & Engineering in the year 2020 in partial fulfilment of the requirements for the award of Degree of Bachelor of Technology in Computer Science & Engineering of Cochin University of Science & Technology.

Dr. Sudheep Elayidom	**Mrs.Sheena S**	**Dr.Latha R Nair**
Project Guide	Project Coordinator	Head of the Divisoin

Kochi
14-04-2020

Mini Project Report

on

Vehicle Entry Monitoring System

Authors:
Naveen Kamal P V
Prakhalbh Santhosh M
Sanjay K Shaji
Sidharth Sunil

Mentor:
Dr.Sudheep Elayidom

Bachelor of Technology
in
Computer Science and Engineering

DIVISION OF COMPUTER ENGINEERING
SCHOOL OF ENGINEERING
COCHIN UNIVERSITY OF SCIENCE & TECHNOLOGY
KOCHI - 682022
April 2020

Acknowledgement

The success and final outcome of this assignment required a lot of guidance and assistance from many people and we extremely fortunate to have got this all along the completion of our work. Firstly, We would like to thank God Almighty for making this project successful. We would like to express our special gratitude to the Head of Department, Computer Science and Engineering **Dr.Latha R Nair** who gave us the golden opportunity to do this wonderful project on the topic Vehicle Entry Monitoring System , which also helped us in doing a lot of Research. We are highly indebted to our mentor **Dr. Sudheep Elayidom** for his guidance and constant supervision as well as for providing necessary information regarding the project and also for the support in completing the project. We would also like to thank faculties **Mrs. Sheena S** and **Mr. Rajin Rayaroth** for their feedback and suggestions which helped immensely in the development of this project.

Last but not least, we would like to express our gratitude to our friends and respondents for the support and willingness to spend some time with us and helped a lot in finalizing this project.

Abstract

Automatic Number Plate Recognition (ANPR) is a special form of Optical Character Recognition (OCR). ANPR is an image processing technology which identifies the vehicle from its number plate automatically by digital pictures. In this project we have presented a model for vehicle number identification based on Optical Character Recognition (OCR). OCR is used to recognize an optically processed printed character number plate which is based on Convolution Neural Network (CNN). This model is tested on different ambient illumination vehicle images. OCR is the last stage in vehicle number plate recognition. In recognition stage the characters on the number plate are converted into texts. The characters are then recognized using the CNN.

The detection and recognition of LPs is done under different conditions and several climatic variations. For that, we present in this project an automatic system for LP detection and recognition based on deep learning approach, which is divided into three parts: detection, segmentation, and character recognition. To detect an LP, many pre-treatment steps should be made before applying the first Convolutional Neural Network (CNN) model for the classification of plates / non-plates. Subsequently, we apply a few pre-processing steps to segment the LP and finally to recognize all the characters in upper case format (A-Z) and digits (0-9), using a second CNN model.

Further we created a database involving obtained License Plate number and entry time of that vehicle.

List of Figures

Contents

1 Introduction

The Automatic License plate recognition (ALPR) is a mass surveillance method that uses optical character recognition on images to read the license plates on vehicles. It has been a frequent topic of research due to many practical applications, such as automatic toll collection, traffic law enforcement, private spaces access control and road traffic monitoring.They can use existing closed-circuit television or road-rule enforcement cameras, or ones specifically designed for the task.

ALPR algorithms are generally divided in four steps: (1) Vehicle image capture (2) Number plate detection (3) Character segmentation and (4) Character recognition. As it is shown in Fig.1, the first step i.e. to capture image of vehicle looks very easy but it is quite exigent task as it is very difficult to capture image of moving vehicle in real time in such a manner that none of the component of vehicle especially the vehicle number plate should be missed. The earlier stages require higher accuracy or almost perfection, since failing to detect the LP would probably lead to a failure in the next stages either. The success of fourth step depends on how second and third step are able to locate vehicle number plate and separate each character. These systems follow different approaches to locate vehicle number plate from vehicle and then to extract vehicle number from that image.

We have utilized You Only Look Once (YOLOv3) object detector to capture and localize the license plate and Optical Character Recognition using Convolutional Neural Networks (CNN) to recognize the license plate. Proposed approaches work end-to end in terms of license plate region detection and license plate recognition.The objective of this project is to successfully locate standard Indian number plate, segment characters and recognize them from a video.

2 Theoretical Background

2.1 You Only Look Once (YOLO) Object Detection Algorithm

You only look once, or YOLO, is one of the faster object detection algorithms.It is a very good choice when you need real-time detection, without loss of too much accuracy. Object detection is a task in computer vision that involves identifying the presence, location, and type of one or more objects in a given photograph.

It is a challenging problem that involves building upon methods for

object recognition (e.g. where are they), object localization (e.g. what are their extent), and object classification (e.g. what are they).

In recent years, deep learning techniques are achieving state-of-the-art results for object detection, such as on standard benchmark datasets and in computer vision competitions. Notable is the "You Only Look Once," or YOLO, family of Convolutional Neural Networks that achieve near state-of-the-art results with a single end-to-end model that can perform object detection in real-time.

How It Works Prior detection systems repurpose classifiers or localizers to perform detection. They apply the model to an image at multiple locations and scales. High scoring regions of the image are considered detections.

We use a totally different approach. We apply a single neural network to the full image. This network divides the image into regions and predicts bounding boxes and probabilities for each region. These bounding boxes are weighted by the predicted probabilities.

Our model has several advantages over classifier-based systems. It looks at the whole image at test time so its predictions are informed by global context in the image. It also makes predictions with a single network evaluation unlike systems like R-CNN which require thousands for a single image. This makes it extremely fast, more than 1000x faster than R-CNN and 100x faster than Fast R-CNN. See our paper for more details on the full system.

Detection at three Scales The newer architecture boasts of residual skip connections, and upsampling. The most salient feature of v3 is that it makes detections at three different scales. YOLO is a fully convolutional network and its eventual output is generated by applying a 1 x 1 kernel on a The shape of the detection kernel is 1 x 1 x (B x (5 + C)). Here B is the number of bounding boxes a cell on the feature map can predict, "5" is for the 4 bounding box attributes and one object confidence, and C is the number of classes. In YOLO v3 trained on COCO, B = 3 and C = 80, so the kernel size is 1 x 1 x 255. The feature map produced by this kernel has identical height and width of the previous feature map, and has detection attributes along the depth as described above. feature map. In YOLO v3, the detection is done by applying 1 x 1 detection kernels on feature maps of three different sizes at three different places in the network. YOLO v3 makes prediction at three scales, which are precisely given by downsampling the dimensions of the input image by 32, 16 and 8 respectively. The first detection is made by the 82nd layer. For the first 81 layers, the image is down sampled by the network, such that the 81st layer has a stride of 32. If we have an image of 416 x 416, the resultant feature map would be of size 13 x 13. One detection is made here using the 1 x 1 detection kernel, giving us a detection feature map of 13 x 13 x 255. Then, the feature map from layer 79 is subjected to a few convolutional layers before being up sampled

by 2x to dimensions of 26 x 26. This feature map is then depth concatenated with the feature map from layer 61. Then the combined feature maps is again subjected a few 1 x 1 convolutional layers to fuse the features from the earlier layer (61). Then, the second detection is made by the 94th layer, yielding a detection feature map of 26 x 26 x 255. A similar procedure is followed again, where the feature map from layer 91 is subjected to few convolutional layers before being depth concatenated with a feature map from layer 36. Like before, a few 1 x 1 convolutional layers follow to fuse the information from the previous layer (36). We make the final of the 3 at 106th layer, yielding feature map of size 52 x 52 x 255. Better at detecting smaller objects Detections at different layers helps address the issue of detecting small objects, a frequent complaint with YOLO v2. The upsampled layers concatenated with the previous layers help preserve the fine grained features which help in detecting small objects. The 13 x 13 layer is responsible for detecting large objects, whereas the 52 x 52 layer detects the smaller objects, with the 26 x 26 layer detecting medium objects. Here is a comparative analysis of different objects picked in the same object by different layers. Choice of anchor boxes YOLO v3, in total uses 9 anchor boxes. Three for each scale. If you're training YOLO on your own dataset, you should go about using K-Means clustering to generate 9 anchors. Then, arrange the anchors is descending order of a dimension. Assign the three biggest anchors for the first scale , the next three for the second scale, and the last three for the third.More bounding boxes per image For an input image of same size, YOLO v3 predicts more bounding boxes than YOLO v2. For instance, at it's native resolution of 416 x 416, YOLO v2 predicted 13 x 13 x 5 = 845 boxes. At each grid cell, 5 boxes were detected using 5 anchors. On the other hand YOLO v3 predicts boxes at 3 different scales. For the same image of 416 x 416, the number of predicted boxes are 10,647. This means that YOLO v3 predicts 10x the number of boxes predicted by YOLO v2. You could easily imagine why it's slower than YOLO v2. At each scale, every grid can predict 3 boxes using 3 anchors. Since there are three scales, the number of anchor boxes used in total are 9, 3 for each scale.

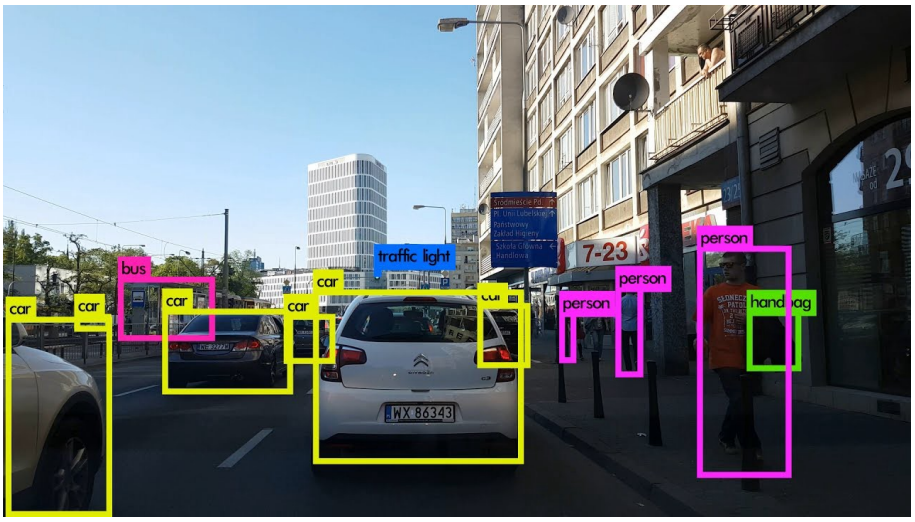

Figure 1: Yolo Example

2.2 Convolutional Neural Network

A Convolutional Neural Network (CNN) is a multilayered neural network with a special architecture to detect complex features in data. CNNs have been used in image recognition, powering vision in robots, and for self-driving vehicles.Convolutional neural networks (CNNs) are primarily supervised deep neural networks used for classification of images, object recognitions and image segmentation applications. More recently, CNNs are also applied to applications involving sound, texts etc.

CNNs are successful models because of its ability to recognize the complex patterns present in images without any hard coding of details or features. The CNN models uses the concept of receptive field and weight sharing, which not only reduces the training parameters but also reduces the complexity of the overall model. The feature map of each layer is generated from the previous layer's receptive field which makes it more suitable for pattern recognition. Normally, a CNN model consists of an input layer, followed by few alternating layers of convolution, activation and pooling and then followed by the fully connected network layers at the very end of the neural network.

Once a CNN is built, it can be used to classify the contents of different images. All we have to do is feed those images into the model. Just like ANNs, CNNs are inspired by the workings of the human brain. CNNs are able to classify images by detecting features, similar to how the human brain detects features to identify objects.

Here we are going to use the CNN model capable of classifying images. An image classifier CNN can be used in many ways, to classify cats and dogs, for example, or to detect if pictures of the brain contain a tumor.

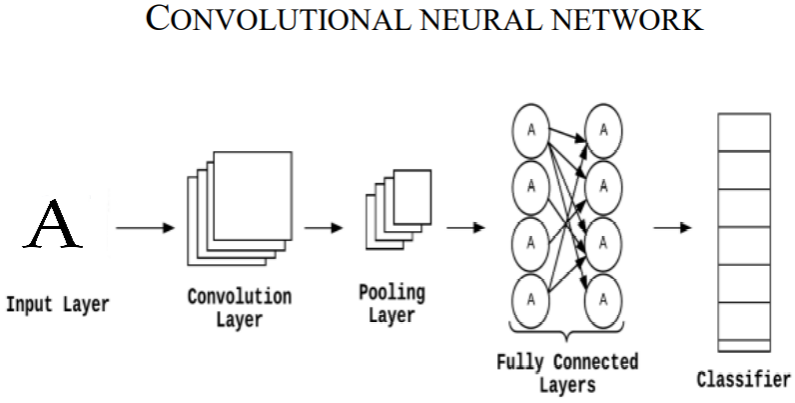

Figure 2: Overview of a Convolutional Neural Network

2.2.1 Working of CNN

Images are made up of pixels. Each pixel is represented by a number between 0 and 255. Therefore each image has a digital representation which is how computers are able to work with images.

1. *Convolution*

A convolution is a combined integration of two functions that shows you how one function modifies the other.

$$(f * g)(t) \stackrel{\text{def}}{=} \int_{-\infty}^{\infty} f(\tau)g(t-\tau)\,d\tau$$
$$= \int_{-\infty}^{\infty} f(t-\tau)g(\tau)\,d\tau.$$

Figure 3: The Convolution Function

There are three important items to mention in this process: the input image, the feature detector, and the feature map. The input image is the image

being detected. The feature detector is a matrix, usually 3x3 (it could also be 7x7). A feature detector is also referred to as a kernel or a filter.

Intuitively, the matrix representation of the input image is multiplied element-wise with the feature detector to produce a feature map, also known as a convolved feature or an activation map. The aim of this step is to reduce the size of the image and make processing faster and easier. Some of the features of the image are lost in this step.

However, the main features of the image that are important in image detection are retained. These features are the ones that are unique to identifying that specific object.

2. *Activation Layer*

In this step we apply the rectifier function to increase non-linearity in the CNN. The commonly used activations functions are Rectified Linear Unit (ReLU), Softmax, Sigmoid and Tanh. For this network, ReLU as in (1) is used as an activation function Images are made of different objects that are not linear to each other. Without applying this function the image classification will be treated as a linear problem while it is actually a non-linear one.

ReLU Activation function,

$$f(x) = max(0, x) \tag{1}$$

Softmax activation function,

$$f(xi) = \frac{Exp(xi)}{\sum_{n}^{j=0} Exp(xj)}(2) \tag{2}$$

where i = 0, 1, 2, ... n 3. *Pooling*

Pooling enables the CNN to detect features in various images irrespective of the difference in lighting in the pictures and different angles of the images.

There are different types of pooling, for example, max pooling and min pooling. Max pooling works by placing a matrix of 2x2 on the feature map and picking the largest value in that box. The 2x2 matrix is moved from left to right through the entire feature map picking the largest value in each pass.

These values then form a new matrix called a pooled feature map. Max pooling works to preserve the main features while also reducing the size of the image. This helps reduce overfitting, which would occur if the CNN is given too

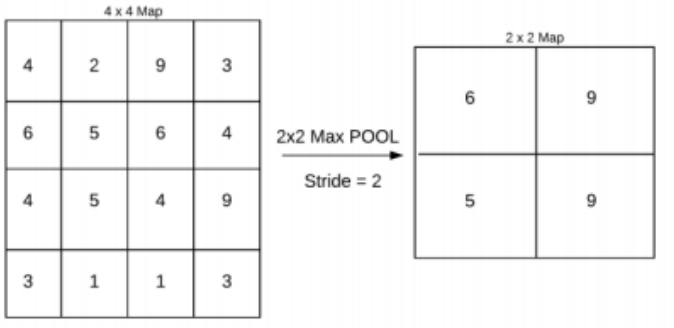

Fig.4. Max Pooling Operation

much information, especially if that information is not relevant in classifying the image.

4. *Flattening*

Once the pooled featured map is obtained, the next step is to flatten it. Flattening involves transforming the entire pooled feature map matrix into a single column which is then fed to the neural network for processing.

5. *Full Connection*

After flattening, the flattened feature map is passed through a neural network.Fully Connected layers are present at the end of the convolutional neural network, their work is to perform classification based on the feature tiles extracted by the convolutional layers. This step is made up of the input layer, the fully connected layer, and the output layer. The output layer is where we get the predicted classes. The information is passed through the network and the error of prediction is calculated. The error is then backpropagated through the system to improve the prediction.

3 Methodology

In this section, the details of the proposed solution for license plate recognition are described. The Proposed solution consist of four parts; Vehicle image capture, License plate detection, character segmentation and license plate recognition. Figure 1 illustrates these parts. In this study, we utilize YOLOv3, which is a popular deep learning based object detection technique, for our object detection purposes. Yolov3 model searches objects in feature maps from various layers that makes it able to detect various sized objects. Using the YOLOv3 object detector, we perform license plate region and character detection as ex-

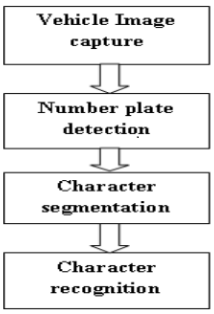

Figure 4: Conventional ALPR system

plained next.

3.1 Vehicle Image Capture and Number Plate Detection

This is the first step toward an ALPR system. We have made use of You Only Look Once(YOLO) for the first and second step. All the details are mentioned below in section 2.3.

3.2 Dataset Collection and Annotation

3.2.1 Dataset Collection

For training, 3244 images of cars with clear licenseplate are fetched from different sources and stored. Two wheeler images was difficult to collect and only 195 images are collected for that category.

3.2.2 Dataset Annotation

There are several tools that can be used to create the annotations for each one of the images that will be part of the training set. This means, to manually indicate the "bounding box" containing each one of the objects in the image and indicate to which class the object belongs. After collecting the required images, the License plate are then marked with a bounding box. The normalized coordinates of the bounding boxes are the stored into each text files with same name as that of the image. Annotations are done using the LabelImg tool `https://github.com/tzutalin/labelImg`.

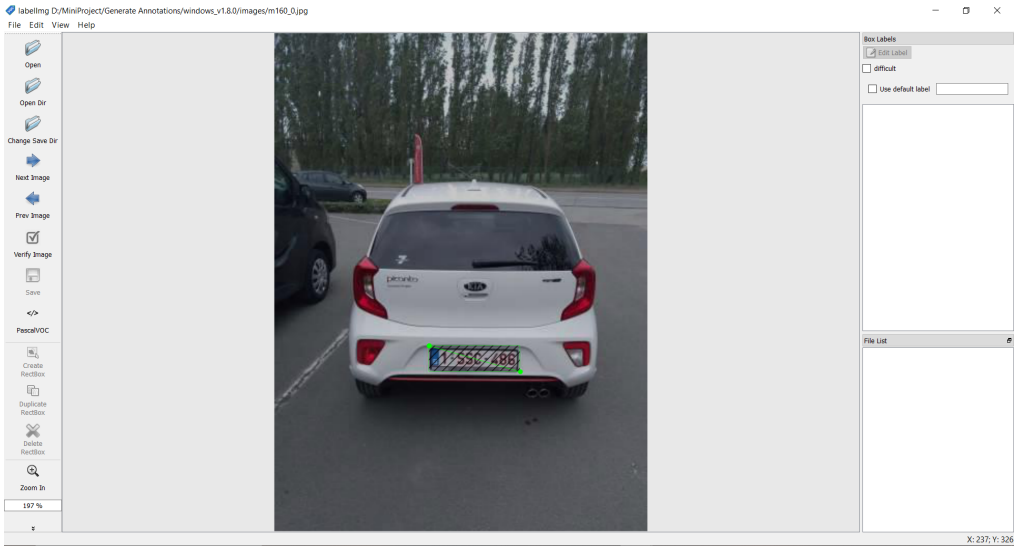

Figure 5: Annotating the images

3.3 Implementation of You Only Look Once (YOLO) Object Detection Algorithm

3.3.1 Creation of Layers of YOLO

Creating the layers of the network architecture. We will have to create a cfg layer according to the number of classes we have to detect[Here= 1].

In configuration there is:

```
[convolutional]
batch=64
subdivisions=16
width=416
height=416
channels=3
momentum=0.9
decay=0.0005
angle=0
saturation = 1.5
exposure = 1.5
hue=.1

[convolutional]
size=1
stride=1
pad=1
filters=18
activation=linear
```

```
20
21 [yolo]
22 mask = 6,7,8
23 anchors = 10,13,  16,30,  33,23,  30,61,  62,45,  59,119,  116,90,
       156,198,  373,326
24 classes=1
25 num=9
26 jitter=.3
27 ignore_thresh = .7
28 truth_thresh = 1
29 random=1
```

**Convolutional layer

```
1 [convolutional]
2 batch_normalize=1
3 size=3
4 stride=1
5 pad=1
6 filters=1024
7 activation=leaky
```

**Shortcut layer

```
1 [shortcut]
2 from=-3
3 activation=linear
```

A shortcut layer is a skip connection, akin to the one used in ResNet. The from parameter is -3, which means the output of the shortcut layer is obtained by adding feature maps from the previous and the 3rd layer backwards from the shortcut layer.

**Upsample layer

```
1 [upsample]
2 stride=2
3 Upsamples the feature map in the previous layer by a factor of
      stride using bilinear upsampling.
```

**Route layer

```
1 [route]
2 layers = -4
3
4 [route]
5 layers = -1, 61
```

The route layer deserves a bit of explanation. It has an attribute layers which can have either one, or two values.

When layers attribute has only one value, it outputs the feature maps of the layer indexed by the value. In our example, it is -4, so the layer will output feature map from the 4th layer backwards from the Route layer.

When layers has two values, it returns the concatenated feature maps of the layers indexed by it's values. In our example it is -1, 61, and the layer will output feature maps from the previous layer (-1) and the 61st layer, concatenated along the depth dimension.

The anchors describes 9 anchors, but only the anchors which are indexed by attributes of the mask tag are used. Here, the value of mask is 0,1,2, which means the first, second and third anchors are used. This make sense since each cell of the detection layer predicts 3 boxes. In total, we have detection layers at 3 scales, making up for a total of 9 anchors. Net There's another type of block called net in the cfg.it only describes information about the network input and training parameters. It isn't used in the forward pass of YOLO. However, it does provide us with information like the network input size, which we use to adjust anchors in the forward pass.

```
[net]
# Testing
batch=1
subdivisions=1
# Training
# batch=64
# subdivisions=16
width= 320
height = 320
channels=3
momentum=0.9
decay=0.0005
angle=0
saturation = 1.5
exposure = 1.5
hue=.1\\
```

	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
1×	Convolutional	32	1 × 1	
	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
2×	Convolutional	64	1 × 1	
	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
8×	Convolutional	128	1 × 1	
	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
8×	Convolutional	256	1 × 1	
	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
4×	Convolutional	512	1 × 1	
	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

Figure 6: 53 convolutional layers called Darknet 53

3.3.2 Parsing the configuration file

The idea here is to parse the cfg, and store every block as a dict. The attributes of the blocks and their values are stored as key-value pairs in the dictionary. As we parse through the cfg, we keep appending these dicts, denoted by the variable block in our code, to a list blocks. Our function will return this block. We are saving the content of configuration file in a list of strings and also performs some preprocessing.

3.3.3 Creating the building blocks

Now we are going to use the list returned by the above parsecfg to construct PyTorch modules for the blocks . We have 5 types of layers in the list . PyTorch provides pre-built layers for types convolutional and upsample.

Before we iterate over list of blocks, we define a variable netinfo to store information about the network.

nn.ModuleList Our function will return a nn.ModuleList. This class is almost like a normal list containing nn.Module objects. However, when we add nn.ModuleList as a member of a nn.Module object (i.e. when we add modules to our network), all the parameters of nn.Module objects (modules) inside the nn.ModuleList are added as parameters of the nn.Module object (i.e.

our network, which we are adding the nn.ModuleList as a member of) as well.

When we define a new convolutional layer, we must define the dimension of it's kernel. While the height and width of kernel is provided by the cfg file, the depth of the kernel is precisely the number of filters (or depth of the feature map) present in the previous layer. This means we need to keep track of number of filters in the layer on which the convolutional layer is being applied. We use the variable prevfilter to do this. We initialise this to 3, as the image has 3 filters corresponding to the RGB channels.

The route layer brings (possibly concatenated) feature maps from previous layers. If there's a convolutional layer right in front of a route layer, then the kernel is applied on the feature maps of previous layers, precisely the ones the route layer brings. Therefore, we need to keep a track of the number of filters in not only the previous layer, but each one of the preceding layers. As we iterate, we append the number of output filters of each block to the list outputfilters.

Now, the idea is to iterate over the list of blocks, and create a PyTorch module for each block as we go.

```
for index, x in enumerate(blocks[1:]):
    module = nn.Sequential()

    #check the type of block
    #create a new module for the block
    #append to module_list
```

nn.Sequential class is used to sequentially execute a number of nn.Module objects. If you look at the cfg, you will realize a block may contain more than one layer. For example, a block of type convolutional has a batch norm layer as well as leaky ReLU activation layer in addition to a convolutional layer. We string together these layers using the nn.Sequential and it's the add_module function. For example, this is how we create the convolutional and the upsample layers.

```
if (x["type"] == "convolutional"):
        #Get the info about the layer
        activation = x["activation"]
        try:
            batch_normalize = int(x["batch_normalize"])
            bias = False
        except:
            batch_normalize = 0
            bias = True

        filters= int(x["filters"])
        padding = int(x["pad"])
        kernel_size = int(x["size"])
        stride = int(x["stride"])

        if padding:
            pad = (kernel_size - 1) // 2
```

```
            else:
                pad = 0

            #Add the convolutional layer
            conv = nn.Conv2d(prev_filters, filters, kernel_size,
    stride, pad, bias = bias)
            module.add_module("conv_{0}".format(index), conv)

            #Add the Batch Norm Layer
            if batch_normalize:
                bn = nn.BatchNorm2d(filters)
                module.add_module("batch_norm_{0}".format(index),
    bn)

            #Check the activation.
            #It is either Linear or a Leaky ReLU for YOLO
            if activation == "leaky":
                activn = nn.LeakyReLU(0.1, inplace = True)
                module.add_module("leaky_{0}".format(index), activn
    )

        #If it's an upsampling layer
        #We use Bilinear2dUpsampling
        elif (x["type"] == "upsample"):
            stride = int(x["stride"])
            upsample = nn.Upsample(scale_factor = 2, mode = "
    bilinear")
            module.add_module("upsample_{}".format(index), upsample
    )
```

Route Layer / Shortcut Layers

Next, we write the code for creating the Route and the Shortcut Layers.

```
        #If it is a route layer
        elif (x["type"] == "route"):
            x["layers"] = x["layers"].split(',')
            #Start  of a route
            start = int(x["layers"][0])
            #end, if there exists one.
            try:
                end = int(x["layers"][1])
            except:
                end = 0
            #Positive anotation
            if start > 0:
                start = start - index
            if end > 0:
                end = end - index
            route = EmptyLayer()
            module.add_module("route_{0}".format(index), route)
            if end < 0:
                filters = output_filters[index + start] +
    output_filters[index + end]
            else:
                filters= output_filters[index + start]

        #shortcut corresponds to skip connection
```

```
        elif x["type"] == "shortcut":
            shortcut = EmptyLayer()
            module.add_module("shortcut_{}".format(index), shortcut
    )
```

At first, we extract the the value of the layers attribute, cast it into an integer and store it in a list.

Then we have a new layer called EmptyLayer which, as the name suggests is just an empty layer.

```
route = EmptyLayer()
```

It is defined as.

```
class EmptyLayer(nn.Module):
    def __init__(self):
        super(EmptyLayer, self).__init__()
```

The convolutional layer just in front of a route layer applies it's kernel to (possibly concatenated) feature maps from a previous layers. The following code updates the filters variable to hold the number of filters outputted by a route layer.

3.3.4 Creating the YOLO Layer

Finally, we write the code for creating the the YOLO layer.

```
        #Yolo is the detection layer
        elif x["type"] == "yolo":
            mask = x["mask"].split(",")
            mask = [int(x) for x in mask]

            anchors = x["anchors"].split(",")
            anchors = [int(a) for a in anchors]
            anchors = [(anchors[i], anchors[i+1]) for i in range(0,
     len(anchors),2)]
            anchors = [anchors[i] for i in mask]

            detection = DetectionLayer(anchors)
            module.add_module("Detection_{}".format(index),
    detection)
```

3.3.5 Detection Layer

We define a new layer Detection Layer that holds the anchors used to detect bounding boxes.

The detection layer is defined as

```
class DetectionLayer(nn.Module):
    def __init__(self, anchors):
        super(DetectionLayer, self).__init__()
        self.anchors = anchors
```

At the end of the loop, we do some bookkeeping.

```
        module_list.append(module)
        prev_filters = filters
        output_filters.append(filters)
```

That concludes the body of the loop. At the end of the function create_modules, we return a tuple containing the net_info, and module_list.

```
  (9): Sequential(
    (conv_9): Conv2d (128, 64, kernel_size=(1, 1), stride=(1, 1),
   bias=False)
    (batch_norm_9): BatchNorm2d(64, eps=1e-05, momentum=0.1,
   affine=True)
    (leaky_9): LeakyReLU(0.1, inplace)
  )
  (10): Sequential(
    (conv_10): Conv2d (64, 128, kernel_size=(3, 3), stride=(1, 1),
    padding=(1, 1), bias=False)
    (batch_norm_10): BatchNorm2d(128, eps=1e-05, momentum=0.1,
   affine=True)
    (leaky_10): LeakyReLU(0.1, inplace)
  )
  (11): Sequential(
    (shortcut_11): EmptyLayer(
    )
  )
.
.
.
```

3.3.6 Defining The Network

```
class Darknet(nn.Module):
    def __init__(self, cfgfile):
        super(Darknet, self).__init__()
        self.blocks = parse_cfg(cfgfile)
        self.net_info, self.module_list = create_modules(self.
    blocks)
```

Implementing the forward pass of the network

```
class Darknet(nn.Module):
def forward(self, x, CUDA):
    modules = self.blocks[1:]
    outputs = {}   #We cache the outputs for the route layer
```

Since route and shortcut layers need output maps from previous layers, we cache the output feature maps of every layer in a dict outputs. The keys are the the indices of the layers, and the values are the feature maps.The thing to notice here is that the modules have been appended in the same order as they are present in the configuration file. This means, we can simply run our input through each module to get our output.

```
class Darknet(nn.Module):
    write = 0     #This is explained a bit later
for i, module in enumerate(modules):
    module_type = (module["type"])
Convolutional and Upsample Layers
```

Convolutional and Upsample Layers
If the module is a convolutional or upsample module, this is how the forward pass should work.

```
        if module_type == "convolutional" or module_type == "
    upsample":
            x = self.module_list[i](x)
```

Route Layer / Shortcut Layer
If you look the code for route layer, we have to account for two cases (as described in part 2). For the case in which we have to concatenate two feature maps we use the torch.cat function with the second argument as 1. This is because we want to concatenate the feature maps along the depth. (In PyTorch, input and output of a convolutional layer has the format 'B X C X H X W. The depth corresponding the the channel dimension).

```
elif module_type == "route":
            layers = module["layers"]
            layers = [int(a) for a in layers]

            if (layers[0]) > 0:
                layers[0] = layers[0] - i

            if len(layers) == 1:
                x = outputs[i + (layers[0])]

            else:
                if (layers[1]) > 0:
                    layers[1] = layers[1] - i

                map1 = outputs[i + layers[0]]
                map2 = outputs[i + layers[1]]

                x = torch.cat((map1, map2), 1)

        elif  module_type == "shortcut":
            from_ = int(module["from"])
            x = outputs[i-1] + outputs[i+from_]
```

YOLO (Detection Layer)

The output of YOLO is a convolutional feature map that contains the bounding box attributes along the depth of the feature map. The attributes bounding boxes predicted by a cell are stacked one by one along each other. So, if you have to access the second bounding of cell at (5,6), then you will have to index it by map[5,6, (5+C): 2*(5+C)]. This form is very inconvenient for output processing such as thresholding by a object confidence, adding grid offsets to centers, applying anchors etc.

Another problem is that since detections happen at three scales, the dimensions of the prediction maps will be different. Although the dimensions of the three feature maps are different, the output processing operations to be done on them are similar. It would be nice to have to do these operations on a single tensor, rather than three separate tensors.

3.3.7 Transforming the output

Predict transform takes in 5 parameters.

```
def predict_transform(prediction, inp_dim, anchors, num_classes,
    CUDA = True):
```

predict_transform function takes an detection feature map and turns it into a 2-D tensor, where each row of the tensor corresponds to attributes of a bounding box Here's the code to do the above transformation. num_classes, and an optional CUDA flag

```
    batch_size = prediction.size(0)
    stride =  inp_dim // prediction.size(2)
    grid_size = inp_dim // stride
    bbox_attrs = 5 + num_classes
    num_anchors = len(anchors)

    prediction = prediction.view(batch_size, bbox_attrs*num_anchors
    , grid_size*grid_size)
    prediction = prediction.transpose(1,2).contiguous()
    prediction = prediction.view(batch_size, grid_size*grid_size*
    num_anchors, bbox_attrs)
```

The dimensions of the anchors are in accordance to the height and width attributes of the net block. These attributes describe the dimensions of the input image, which is larger (by a factor of stride) than the detection map.

```
anchors = [(a[0]/stride, a[1]/stride) for a in anchors]
```

Now, we need to transform our output according to the equations Sigmoid the x,y coordinates and the objectness score.

```
    #Sigmoid the  centre_X, centre_Y. and object confidencce
    prediction[:,:,0] = torch.sigmoid(prediction[:,:,0])
    prediction[:,:,1] = torch.sigmoid(prediction[:,:,1])
    prediction[:,:,4] = torch.sigmoid(prediction[:,:,4])
```

Add the grid offsets to the center cordinates prediction.

```
    #Add the center offsets
    grid = np.arange(grid_size)
    a,b = np.meshgrid(grid, grid)

    x_offset = torch.FloatTensor(a).view(-1,1)
    y_offset = torch.FloatTensor(b).view(-1,1)

    if CUDA:
        x_offset = x_offset.cuda()
        y_offset = y_offset.cuda()

    x_y_offset = torch.cat((x_offset, y_offset), 1).repeat(1,
    num_anchors).view(-1,2).unsqueeze(0)

    prediction[:,:,:2] += x_y_offset
```

Apply the anchors to the dimensions of the bounding box.

```
    #log space transform height and the width
    anchors = torch.FloatTensor(anchors)

    if CUDA:
        anchors = anchors.cuda()

    anchors = anchors.repeat(grid_size*grid_size, 1).unsqueeze(0)
    prediction[:,:,2:4] = torch.exp(prediction[:,:,2:4])*anchors
```

Apply sigmoid activation to the the class scores

```
    prediction[:,:,5: 5 + num_classes] = torch.sigmoid((prediction
    [:,:, 5 : 5 + num_classes]))
```

The last thing we want to do here, is to resize the detections map to the size of the input image. The bounding box attributes here are sized according to the feature map (say, 13 x 13). If the input image was 416 x 416, we multiply the attributes by 32, or the stride variable.

```
   prediction[:,:,:4] *= stride
```

3.3.8 Testing the forward pass

Here's a function that creates a dummy input. We will pass this input to our network. Before we write this function, save this image into your working directory .
Forward function.

```
        elif module_type == 'yolo':

            anchors = self.module_list[i][0].anchors
            #Get the input dimensions
```

```
            inp_dim = int (self.net_info["height"])

            #Get the number of classes
            num_classes = int (module["classes"])

            #Transform
            x = x.data
            x = predict_transform(x, inp_dim, anchors, num_classes,
     CUDA)
            if not write:              #if no collector has been
    intialised.
                detections = x
                write = 1

            else:
                detections = torch.cat((detections, x), 1)

        outputs[i] = x
```

```
def get_test_input():
    img = cv2.imread("example.png")
    img = cv2.resize(img, (416,416))
    img_ =  img[:,:,::-1].transpose((2,0,1))
    img_ = img_[np.newaxis,:,:,:]/255.0
    img_ = torch.from_numpy(img_).float()
    img_ = Variable(img_)
    return img_
```

3.3.9 Downloading the Pre-trained Weights

Download the weights file into your detector directory. Understanding the Weights File The official weights file is binary file that contains weights stored in a serial fashion.

Extreme care must be taken to read the weights. The weights are just stored as floats, with nothing to guide us as to which layer do they belong to. If you screw up, there's nothing stopping you to, say, load the weights of a batch norm layer into those of a convolutional layer. First, the weights belong to only two types of layers, either a batch norm layer or a convolutional layer.

The weights for these layers are stored exactly in the same order as they appear in the configuration file. So, if a convolutional is followed by a shortcut block, and then the shortcut block by another convolutional block, You will expect file to contain the weights of the previous convolutional block, followed by those of the latter.

When the batch norm layer appears in a convolutional block, there are no biases. However, when there's no batch norm layer, bias "weights" have to read from the file.

Loading Weights

Let us write a function load weights. It will be a member function of the Darknet class. It'll take one argument other than self, the path of the weightsfile.

```
    def load_weights(self, weightfile):
```

The first 160 bytes of the weights file store 5 int32 values which constitute the header of the file.

```

    #Open the weights file
    fp = open(weightfile, "rb")

    #The first 5 values are header information
    # 1. Major version number
    # 2. Minor Version Number
    # 3. Subversion number
    # 4,5. Images seen by the network (during training)
    header = np.fromfile(fp, dtype = np.int32, count = 5)
    self.header = torch.from_numpy(header)
    self.seen = self.header[3]
```

The rest of bits now represent the weights, in the order described above. The weights are stored as float32 or 32-bit floats.

```
    weights = np.fromfile(fp, dtype = np.float32)
```

Now, we iterate over the weights file, and load the weights into the modules of our network.

```
    ptr = 0
    for i in range(len(self.module_list)):
        module_type = self.blocks[i + 1]["type"]

        #If module_type is convolutional load weights
        #Otherwise ignore.
```

Into the loop, we first check whether the convolutional block has batch normalise True or not. Based on that, we load the weights.

```
        if module_type == "convolutional":
            model = self.module_list[i]
            try:
                batch_normalize = int(self.blocks[i+1]["
    batch_normalize"])
            except:
                batch_normalize = 0
                  conv = model[0]
```

We keep a variable called ptr to keep track of where we are in the weights array. Now, if batch normalize is True, we load the weights as follows.

```
if (batch_normalize):
            bn = model[1]
```

```
3
4               #Get the number of weights of Batch Norm Layer
5               num_bn_biases = bn.bias.numel()
6
7               #Load the weights
8               bn_biases = torch.from_numpy(weights[ptr:ptr +
    num_bn_biases])
9               ptr += num_bn_biases
10
11              bn_weights = torch.from_numpy(weights[ptr: ptr +
    num_bn_biases])
12              ptr  += num_bn_biases
13
14              bn_running_mean = torch.from_numpy(weights[ptr: ptr +
    num_bn_biases])
15              ptr  += num_bn_biases
16
17              bn_running_var = torch.from_numpy(weights[ptr: ptr +
    num_bn_biases])
18              ptr  += num_bn_biases
19
20              #Cast the loaded weights into dims of model weights.
21              bn_biases = bn_biases.view_as(bn.bias.data)
22              bn_weights = bn_weights.view_as(bn.weight.data)
23              bn_running_mean = bn_running_mean.view_as(bn.
    running_mean)
24              bn_running_var = bn_running_var.view_as(bn.running_var)
25
26              #Copy the data to model
27              bn.bias.data.copy_(bn_biases)
28              bn.weight.data.copy_(bn_weights)
29              bn.running_mean.copy_(bn_running_mean)
30              bn.running_var.copy_(bn_running_var)
31  if (batch_normalize):
32              bn = model[1]
33
34              #Get the number of weights of Batch Norm Layer
35              num_bn_biases = bn.bias.numel()
36
37              #Load the weights
38              bn_biases = torch.from_numpy(weights[ptr:ptr +
    num_bn_biases])
39              ptr += num_bn_biases
40
41              bn_weights = torch.from_numpy(weights[ptr: ptr +
    num_bn_biases])
42              ptr  += num_bn_biases
43
44              bn_running_mean = torch.from_numpy(weights[ptr: ptr +
    num_bn_biases])
45              ptr  += num_bn_biases
46
47              bn_running_var = torch.from_numpy(weights[ptr: ptr +
    num_bn_biases])
48              ptr  += num_bn_biases
49
50              #Cast the loaded weights into dims of model weights.
```

```
            bn_biases = bn_biases.view_as(bn.bias.data)
            bn_weights = bn_weights.view_as(bn.weight.data)
            bn_running_mean = bn_running_mean.view_as(bn.
    running_mean)
            bn_running_var = bn_running_var.view_as(bn.running_var)

            #Copy the data to model
            bn.bias.data.copy_(bn_biases)
            bn.weight.data.copy_(bn_weights)
            bn.running_mean.copy_(bn_running_mean)
            bn.running_var.copy_(bn_running_var)
```

If batch_norm is not true, simply load the biases of the convolutional layer.

```

        else:
            #Number of biases
            num_biases = conv.bias.numel()

            #Load the weights
            conv_biases = torch.from_numpy(weights[ptr: ptr +
    num_biases])
            ptr = ptr + num_biases

            #reshape the loaded weights according to the dims of
    the model weights
            conv_biases = conv_biases.view_as(conv.bias.data)

            #Finally copy the data
            conv.bias.data.copy_(conv_biases)
```

Finally, we load the convolutional layer's weights at last.

```
#Let us load the weights for the Convolutional layers
num_weights = conv.weight.numel()

#Do the same as above for weights
conv_weights = torch.from_numpy(weights[ptr:ptr+num_weights])
ptr = ptr + num_weights

conv_weights = conv_weights.view_as(conv.weight.data)
conv.weight.data.copy_(conv_weights)
```

We're done with this function and you can now load weights in your Darknet object by calling the load_weights function on the darknet object. Object Confidence Thresholding Our prediction tensor contains information about B x 10647 bounding boxes. For each of the bounding box having a objectness score below a threshold, we set the values of it's every attribute (entire row representing the bounding box) to zero.

3.3.10 Non-maximum Suppression

Actually, after single forward pass CNN, what's going to happen is the YOLO network is trying to suggest multiple bounding boxes for the same detected object. The problem is how do we decide which one of these bounding boxes is the right one. Fortunately, to overcome this problem, a method called non-maximum suppression (NMS) can be applied. Basically, what NMS does is to clean up these detections. The first step of NMS is to suppress all the predictions boxes where the confidence score is under a certain threshold value. Let's say the confidence threshold is set to 0.5, so every bounding box where the confidence score is less than or equal to 0.5 will be discarded. The bounding box attributes we have now are described by the center coordinates, as well as the height and width of the bounding box. However, it's easier to calculate IoU of two boxes, using coordinates of a pair of diagnal corners of each box. So, we transform the (center x, center y, height, width) attributes of our boxes, to (top-left corner x, top-left corner y, right-bottom corner x, right-bottom corner y).

```
    box_corner = prediction.new(prediction.shape)
    box_corner[:,:,0] = (prediction[:,:,0] - prediction[:,:,2]/2)
    box_corner[:,:,1] = (prediction[:,:,1] - prediction[:,:,3]/2)
    box_corner[:,:,2] = (prediction[:,:,0] + prediction[:,:,2]/2)
    box_corner[:,:,3] = (prediction[:,:,1] + prediction[:,:,3]/2)
    prediction[:,:,:4] = box_corner[:,:,:4]
```

The number of true detections in every image may be different. For example, a batch of size 3 where images 1, 2 and 3 have 5, 2, 4 true detections respectively. Therefore, confidence thresholding and NMS has to be done for one image at once. This means, we cannot vectorise the operations involved, and must loop over the first dimension of prediction (containing indexes of images in a batch).

```
    batch_size = prediction.size(0)

    write = False

    for ind in range(batch_size):
        image_pred = prediction[ind]          #image Tensor
           #confidence threshholding
           #NMS
```

```
max_conf, max_conf_score = torch.max(image_pred[:,5:5+ num_classes
    ], 1)
        max_conf = max_conf.float().unsqueeze(1)
        max_conf_score = max_conf_score.float().unsqueeze(1)
        seq = (image_pred[:,:5], max_conf, max_conf_score)
        image_pred = torch.cat(seq, 1)
```

Remember we had set the bounding box rows having a object confidence less than the threshold to zero.

```
 non_zero_ind =  (torch.nonzero(image_pred[:,4]))
```

```
        try:
            image_pred_ = image_pred[non_zero_ind.squeeze(),:].view
    (-1,7)
        except:
            continue

        #For PyTorch 0.4 compatibility
        #Since the above code with not raise exception for no
    detection
        #as scalars are supported in PyTorch 0.4
        if image_pred_.shape[0] == 0:
            continue
```

Since there can be multiple true detections of the same class, we use a function called unique to get classes present in any given image.

```
def unique(tensor):
    tensor_np = tensor.cpu().numpy()
    unique_np = np.unique(tensor_np)
    unique_tensor = torch.from_numpy(unique_np)

    tensor_res = tensor.new(unique_tensor.shape)
    tensor_res.copy_(unique_tensor)
    return tensor_res
```

Then, we perform NMS classwise.

```
        for cls in img_classes:
            #perform NMS
```

Once we are inside the loop, the first thing we do is extract the detections of a particular class (denoted by variable cls).

```
#get the detections with one particular class
cls_mask = image_pred_*(image_pred_[:,-1] == cls).float().unsqueeze
    (1)
class_mask_ind = torch.nonzero(cls_mask[:,-2]).squeeze()
image_pred_class = image_pred_[class_mask_ind].view(-1,7)

#sort the detections such that the entry with the maximum
    objectness
s#confidence is at the top
conf_sort_index = torch.sort(image_pred_class[:,4], descending =
    True )[1]
image_pred_class = image_pred_class[conf_sort_index]
idx = image_pred_class.size(0)   #Number of detections
```

Now, we perform NMS.

```
for i in range(idx):
    #Get the IOUs of all boxes that come after the one we are
    looking at
    #in the loop
    try:
        ious = bbox_iou(image_pred_class[i].unsqueeze(0),
    image_pred_class[i+1:])
```

```
    except ValueError:
        break

    except IndexError:
        break

    #Zero out all the detections that have IoU > treshhold
    iou_mask = (ious < nms_conf).float().unsqueeze(1)
    image_pred_class[i+1:] *= iou_mask

    #Remove the non-zero entries
    non_zero_ind = torch.nonzero(image_pred_class[:,4]).squeeze()
    image_pred_class = image_pred_class[non_zero_ind].view(-1,7)
```

Here, we use a function bbox_iou. The first input is the bounding box row that is indexed by the the variable i in the loop. Second input to bbox_iou is a tensor of multiple rows of bounding boxes. The output of the function bbox_iou is a tensor containing IoUs of the bounding box represented by the first input with each of the bounding boxes present in the second input. If we have two bounding boxes of the same class having an an IoU larger than a threshold, then the one with lower class confidence is eliminated. We've already sorted out bounding boxes with the ones having higher confidences at top. In the body of the loop, the following lines gives the IoU of box, indexed by i with all the bounding boxes having indices higher than i.

```
ious = bbox_iou(image_pred_class[i].unsqueeze(0), image_pred_class[
    i+1:])
```

Every iteration, if any of the bounding boxes having indices greater than i have an IoU (with box indexed by i) larger than the threshold nms_thresh, than that particular box is eliminated.

```
#Zero out all the detections that have IoU > treshhold
iou_mask = (ious < nms_conf).float().unsqueeze(1)
image_pred_class[i+1:] *= iou_mask

#Remove the non-zero entries
non_zero_ind = torch.nonzero(image_pred_class[:,4]).squeeze()
image_pred_class = image_pred_class[non_zero_ind]
```

We have put the line of code to compute the ious in a try-catch block. This is because the loop is designed to run idx iterations (number of rows in image_pred_class). However, as we proceed with the loop, a number of bounding boxes may be removed from image_pred_class. This means, even if one value is removed from image_pred_class, we cannot have idx iterations. Hence, we might try to index a value that is out of bounds (IndexError), or the slice image_pred_class[i+1:] may return an empty tensor, assigning which triggers a ValueError. At that point, we can ascertain that NMS can remove no further bounding boxes, and we break out of the loop. Calculating the IoU Here is the function bbox_iou.

```
def bbox_iou(box1, box2):
    """
    Returns the IoU of two bounding boxes


    """
    #Get the coordinates of bounding boxes
    b1_x1, b1_y1, b1_x2, b1_y2 = box1[:,0], box1[:,1], box1[:,2], box1[:,3]
    b2_x1, b2_y1, b2_x2, b2_y2 = box2[:,0], box2[:,1], box2[:,2], box2[:,3]

    #get the corrdinates of the intersection rectangle
    inter_rect_x1 =  torch.max(b1_x1, b2_x1)
    inter_rect_y1 =  torch.max(b1_y1, b2_y1)
    inter_rect_x2 =  torch.min(b1_x2, b2_x2)
    inter_rect_y2 =  torch.min(b1_y2, b2_y2)

    #Intersection area
    inter_area = torch.clamp(inter_rect_x2 - inter_rect_x1 + 1, min=0) * torch.clamp(inter_rect_y2 - inter_rect_y1 + 1, min=0)

    #Union Area
    b1_area = (b1_x2 - b1_x1 + 1)*(b1_y2 - b1_y1 + 1)
    b2_area = (b2_x2 - b2_x1 + 1)*(b2_y2 - b2_y1 + 1)

    iou = inter_area / (b1_area + b2_area - inter_area)

    return iou
```

The function write_results outputs a tensor of shape D x 8. Here D is the true detections in all of images, each represented by a row. Each detections has 8 attributes, namely, index of the image in the batch to which the detection belongs to, 4 corner coordinates, objectness score, the score of class with maximum confidence, and the index of that class. Just as before, we do not initialize our output tensor unless we have a detection to assign to it. Once it has been initialized, we concatenate subsequent detections to it. We use the write flag to indicate whether the tensor has been initialized or not. At the end of loop that iterates over classes, we add the resultant detections to the tensor output.

```

            batch_ind = image_pred_class.new(image_pred_class.size(0), 1).fill_(ind)
            #Repeat the batch_id for as many detections of the class cls in the image
            seq = batch_ind, image_pred_class

            if not write:
                output = torch.cat(seq,1)
                write = True
            else:
                out = torch.cat(seq,1)
                output = torch.cat((output,out))
```

At the end of the function, we check whether output has been initialized at all or not. If it hasn't been means there's hasn't been a single detection in any images of the batch. In that case, we return 0.

```

    try:
        return output
    except:
        return 0
```

3.3.11 Loading the Network

Download the file coco.names.File that contains the names of the objects in the COCO dataset. Create a folder data in your detector directory.Then, we load the class file in our program.load_classes is a function defined in util.py that returns a dictionary which maps the index of every class to a string of it's name.

```
def load_classes(namesfile):
    fp = open(namesfile, "r")
    names = fp.read().split("\n")[:-1]
    return names
```

Initialize the network and load weights.

```
#Set up the neural network
print("Loading network.....")
model = Darknet(args.cfgfile)
model.load_weights(args.weightsfile)
print("Network successfully loaded")

model.net_info["height"] = args.reso
inp_dim = int(model.net_info["height"])
assert inp_dim % 32 == 0
assert inp_dim > 32

#If there's a GPU availible, put the model on GPU
if CUDA:
    model.cuda()

#Set the model in evaluation mode
model.eval()
```

Read the Input images Read the image from the disk, or the images from a directory. The paths of the image/images are stored in a list called imlist.

```
read_dir = time.time()
#Detection phase
try:
    imlist = [osp.join(osp.realpath('.'), images, img) for img in
    os.listdir(images)]
except NotADirectoryError:
```

```
    imlist = []
    imlist.append(osp.join(osp.realpath('.'), images))
except FileNotFoundError:
    print ("No file or directory with the name {}".format(images))
    exit()
```

read_dir is a checkpoint used to measure time. (We will encounter several of these) If the directory to save the detections, defined by the det flag, doesn't exist, create it. Python code highlighting

```
if not os.path.exists(args.det):
    os.makedirs(args.det)
```

3.3.12 OpenCV to load the images.

```
load_batch = time.time()
loaded_ims = [cv2.imread(x) for x in imlist]
```

OpenCV loads an image as an numpy array, with BGR as the order of the color channels. PyTorch's image input format is (Batches x Channels x Height x Width), with the channel order being RGB. Therefore, we write the function prep_image in util.py to transform the numpy array into PyTorch's input format. Before we can write this function, we must write a function letterbox_image that resizes our image, keeping the aspect ratio consistent, and padding the left out areas with the color (128,128,128).

```
def letterbox_image(img, inp_dim):
    '''resize image with unchanged aspect ratio using padding'''
    img_w, img_h = img.shape[1], img.shape[0]
    w, h = inp_dim
    new_w = int(img_w * min(w/img_w, h/img_h))
    new_h = int(img_h * min(w/img_w, h/img_h))
    resized_image = cv2.resize(img, (new_w,new_h), interpolation =
    cv2.INTER_CUBIC)

    canvas = np.full((inp_dim[1], inp_dim[0], 3), 128)

    canvas[(h-new_h)//2:(h-new_h)//2 + new_h,(w-new_w)//2:(w-new_w)
    //2 + new_w,  :] = resized_image

    return canvas
```

Now, we write the function that takes a OpenCV images and converts it to the input of our network.

```
def prep_image(img, inp_dim):
    """
    Prepare image for inputting to the neural network.

    Returns a Variable
```

```
    """

    img = cv2.resize(img, (inp_dim, inp_dim))
    img = img[:,:,::-1].transpose((2,0,1)).copy()
    img = torch.from_numpy(img).float().div(255.0).unsqueeze(0)
    return img
```

3.3.13 Create the Batches

```
leftover = 0
if (len(im_dim_list) % batch_size):
   leftover = 1

if batch_size != 1:
   num_batches = len(imlist) // batch_size + leftover
   im_batches = [torch.cat((im_batches[i*batch_size : min((i +  1)*
    batch_size,
                        len(im_batches))]))  for i in range(
    num_batches)]
```

3.3.14 Drawing the Bounding Box

The Detection Loop
We iterate over the batches, generate the prediction, and concatenate the prediction tensors (of shape, D x 8, the output of write_results function) of all the images we have to perform detections upon. For each batch, we measure the time taken for detection as the time spent between taking the input, and producing the output of the write_results function. In the output returned by write_prediction, one of the attributes was the index of the image in batch. We transform that particular attribute in such a way that it now represents the index of the image in imlist, the list containing addresses of all images. After that, we print time taken for each detection as well as the object detected in each image. If the output of the write_results function for batch is an int(0), meaning there is no detection, we use continue to skip the rest loop.

```

write = 0
start_det_loop = time.time()
for i, batch in enumerate(im_batches):
    #load the image
    start = time.time()
    if CUDA:
        batch = batch.cuda()

    prediction = model(Variable(batch, volatile = True), CUDA)

    prediction = write_results(prediction, confidence, num_classes,
     nms_conf = nms_thesh)

    end = time.time()

    if type(prediction) == int:
```

```
17
18          for im_num, image in enumerate(imlist[i*batch_size: min((i
    +  1)*batch_size, len(imlist))]):
19              im_id = i*batch_size + im_num
20              print("{0:20s} predicted in {1:6.3f} seconds".format(
    image.split("/")[-1], (end - start)/batch_size))
21              print("{0:20s} {1:s}".format("Objects Detected:", ""))
22              print("
    ----------------------------------------------------------")
23          continue
24
25      prediction[:,0] += i*batch_size    #transform the atribute from
     index in batch to index in imlist
26
27      if not write:                      #If we have't initialised
    output
28          output = prediction
29          write = 1
30      else:
31          output = torch.cat((output,prediction))
32
33      for im_num, image in enumerate(imlist[i*batch_size: min((i +
    1)*batch_size, len(imlist))]):
34          im_id = i*batch_size + im_num
35          objs = [classes[int(x[-1])] for x in output if int(x[0]) ==
     im_id]
36          print("{0:20s} predicted in {1:6.3f} seconds".format(image.
    split("/")[-1], (end - start)/batch_size))
37          print("{0:20s} {1:s}".format("Objects Detected:", " ".join(
    objs)))
38          print("
    ----------------------------------------------------------")
39
40      if CUDA:
41          torch.cuda.synchronize()
```

The line torch.cuda.synchronize makes sure that CUDA kernel is synchronized with the CPU. Otherwise, CUDA kernel returns the control to CPU as soon as the GPU job is queued and well before the GPU job is completed (Asynchronous calling). This might lead to a misleading time if end = time.time() gets printed before the GPU job is actually over.

Now, we have the detections of all images in our Tensor Output. Let us draw the bounding boxes on images.

Drawing bounding boxes on images, we use a try-catch block to check whether there has been a single detection has been made or not. If that's not the case, exit the program.

```
1
2 try:
3     output
4 except NameError:
5     print ("No detections were made")
6     exit()
```

Before we draw the bounding boxes, the predictions contained in our output tensor conform to the input size of the network, and not the original sizes of the images. So, before we can draw the bounding boxes, let us transform the corner attributes of each bounding box, to the original dimensions of images.

Before we draw the bounding boxes, the predictions contained in our output tensor are predictions on the padded image, and not the original image. Merely, re-scaling them to the dimensions of the input image won't work here. We first need to transform the co-ordinates of the boxes to be measured with respect to boundaries of the area on the padded image that contains the original image.

```
im_dim_list = torch.index_select(im_dim_list, 0, output[:,0].long()
    )

scaling_factor = torch.min(inp_dim/im_dim_list,1)[0].view(-1,1)


output[:,[1,3]] -= (inp_dim - scaling_factor*im_dim_list[:,0].view
    (-1,1))/2
output[:,[2,4]] -= (inp_dim - scaling_factor*im_dim_list[:,1].view
    (-1,1))/2
```

Now, our co-ordinates conform to dimensions of our image on the padded area. However, in the function letterbox_image, we had resized both the dimensions of our image by a scaling factor (remember both dimensions were divided with a common factor to maintain aspect ratio). We now undo this rescaling to get the co-ordinates of the bounding box on the original image.

```
output[:,1:5] /= scaling_factor
```

Let us now clip any bounding boxes that may have boundaries outside the image to the edges of our image.

```
for i in range(output.shape[0]):
    output[i, [1,3]] = torch.clamp(output[i, [1,3]], 0.0,
    im_dim_list[i,0])
    output[i, [2,4]] = torch.clamp(output[i, [2,4]], 0.0,
    im_dim_list[i,1])
```

If there are too many bounding boxes in the image, drawing them all in one color may not be such a nice idea. Download this file to your detector folder. This is a pickled file that contains many colors to randomly choose from.

```
class_load = time.time()
```

Now let us write a function to draw the boxes.

```
draw = time.time()

def write(x, results, color):
    c1 = tuple(x[1:3].int())
```

```
    c2 = tuple(x[3:5].int())
    img = results[int(x[0])]
    cls = int(x[-1])
    label = "{0}".format(classes[cls])
    cv2.rectangle(img, c1, c2,color, 1)
    t_size = cv2.getTextSize(label, cv2.FONT_HERSHEY_PLAIN, 1 , 1)[0]
    c2 = c1[0] + t_size[0] + 3, c1[1] + t_size[1] + 4
    cv2.rectangle(img, c1, c2,color, -1)
    cv2.putText(img, label, (c1[0], c1[1] + t_size[1] + 4), cv2.FONT_HERSHEY_PLAIN, 1, [225,255,255], 1);
    return img
```

$$x_1 = b_x - \frac{w}{2}$$

$$y_1 = b_y - \frac{h}{2}$$

Figure 7: Bounding Box Equation

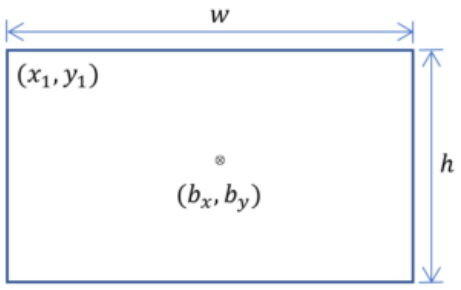

Figure 8: Bounding Box Formation

The function above draws a rectangle with a color of a random choice from colors. It also creates a filled rectangle on the top left corner of the bounding box, and writes the class of the object detected across the filled rectangle. -1 argument of the cv2.rectangle function is used for creating a filled rectangle.

We define write function locally so that it can access the colors list.

We could have also included colors as an argument, but that would have allowed us to use only one color per image, which defeats the purpose of what we want to do.

Once we've defined this function, let us now draw the bounding boxes on images.

```
list(map(lambda x: write(x, loaded_ims), output))
```

The above snippet modifies the images inside loaded_ims inplace. Each image is saved by prefixing the "det_" in front of the image name. We create a list of addresses, to which we will save the our detection images to.

```
det_names = pd.Series(imlist).apply(lambda x: "{}/det_{}".format(
    args.det,x.split("/")[-1]))
```

Finally, write the images with detections to the address in det_names.

```
list(map(cv2.imwrite, det_names, loaded_ims))
end = time.time()
```

3.3.15 Printing Time Summary

At the end of our detector we will print a summary containing which part of the code took how long to execute. This is useful when we have to compare how different hyper-parameters effect the speed of the detector. Hyper-parameters such as batch size, objectness confidence and NMS threshold, (passed with bs, confidence, nms_thresh flags respectively) can be set while executing the script detection.py on the command line.

3.3.16 Running the Detector on Video/Webcam

In order to run the detector on the video or webcam, the code remains almost the same, except we don't have to iterate over batches, but frames of a video.

First, we open the video / camera feed in OpenCV.

```
videofile = "video.avi" #or path to the video file.

cap = cv2.VideoCapture(videofile)

#cap = cv2.VideoCapture(0)  for webcam

assert cap.isOpened(), 'Cannot capture source'

frames = 0
```

Every iteration, we keep a track of the number of frames captured in a variable called frames. We then divide this number by the time elapsed since the first frame to print the FPS of the video.

Instead for writing the detection images to disk using cv2.imwrite, we use cv2.imshow to display the frame with bounding box drawn on it. If the user presses the Q button, it causes the code to break the loop, and the video ends.

```
frames = 0
start = time.time()

while cap.isOpened():
    ret, frame = cap.read()

    if ret:
        img = prep_image(frame, inp_dim)
#        cv2.imshow("a", frame)
        im_dim = frame.shape[1], frame.shape[0]
        im_dim = torch.FloatTensor(im_dim).repeat(1,2)

        if CUDA:
            im_dim = im_dim.cuda()
            img = img.cuda()

        output = model(Variable(img, volatile = True), CUDA)
        output = write_results(output, confidence, num_classes,
    nms_conf = nms_thesh)


        if type(output) == int:
            frames += 1
            print("FPS of the video is {:5.4f}".format( frames / (
    time.time() - start)))
            cv2.imshow("frame", frame)
            key = cv2.waitKey(1)
            if key & 0xFF == ord('q'):
                break
            continue
        output[:,1:5] = torch.clamp(output[:,1:5], 0.0, float(
    inp_dim))

        im_dim = im_dim.repeat(output.size(0), 1)/inp_dim
        output[:,1:5] *= im_dim

        classes = load_classes('data/coco.names')
        colors = pkl.load(open("pallete", "rb"))

        list(map(lambda x: write(x, frame), output))

        cv2.imshow("frame", frame)
        key = cv2.waitKey(1)
        if key & 0xFF == ord('q'):
            break
        frames += 1
        print(time.time() - start)
        print("FPS of the video is {:5.2f}".format( frames / (time.
    time() - start)))
```

```
46      else:
47          break
```

Before Detection

After Detection

Figure 9: Drawing the Bounding Box

3.4 Motion Estimation using Kalman Filtering

Motion estimation is the process of determining motion vectors from a sequence of frames. It has been referred as one of the promising techniques for effective motion estimation and hence numerous variants of this algorithm .

Kalman filter is used to estimate the moving object in the given image sequence. The estimated moving object is used for licence plate detection. Due to the noise and limitation of the detection method, such a central position representation sometimes does not reflect the accurate location of a moving object. When multiple objects occlude, the positions need to be further delineated in order to track each object. Through the tracking approach described in this section, we hope to obtain an optimal estimation of the path of eachobject, hence, to achieve robust tracking of multiple moving objects with occlusion.

Initialization

Select the first image as the reference image and the second image as the current image. Assuming the initial location of the M moving object are L1, L2, ..., LM, respectively. The detection and tracking parts are both initialized.

Step 1 (Detection)

Calculate motion vector of the current image using a block matching algorithm [5]. For the i-th moving object, we search around Li in the current image for the region with non-zero motion vector values, and then calculate the center of the region as the detected location of the moving object.

Step 2 (Tracking)

Pass the location point as the measurement value to the UKF update process to obtain the estimated value of current each moving object's location. The UKF estimates or predicts all the objects simultaneously and independently. The prediction process of UKF is then invoked and the predicted velocity is fed back to the detection step (Step 1).

Step 3

Assign the current image as the reference image and the next image in the input image sequence as the current image; go to Step 1. For the tracking of multiple moving objects, the occlusion among objects is our main concern. If the moving objects are just located at different places and move with no overlap, the case is nearly the same as we detect and track multiple independent objects simultaneously. What we want to tackle in this study is the scenarios where

moving objects are quite close sometimes, where occlusion exits. As such, we apply our detection and tracking scheme to complicated and challenging cases when there are multiple moving objects with occlusion.Based on the tracking approach, i.e., the UKF approach in our case, we can obtain more accurate location information for each person in motion. By considering the continuity of velocity of a moving object, we can match the detected result after occlusion to that before occlusion. This is quite important to resolve the ambiguity existing in most object detection methods.Thus, based on such information, we can clearly predict the next position after occlusion and find the correct one around our predicted position from tracking each object continuously.

3.5 Image preprocessing

This step is essential to enhance the input image and making it more suitable for the next processing steps. The first step done in the preprocessing is to denoise the image. This process is followed by masking to detect black pixels and non-black pixels. The license plate images were found to have non-pure black. So the black pixels taken to consideration were generalized to a range of pixel values than that of pure black. Then the images were smoothened to increase accuracy of prediction.

3.5.1 Denoising

Denoising of an image refers to the process of reconstruction of a signal from noisy images. Denoising is done to remove unwanted noise from image to analyze it in better form. It refers to one of the major pre-processing steps.

Syntax: cv2.fastNlMeansDenoisingColored(P1, P2, float P3, float P4, int P5, int P6)

Parameters: P1 – Source Image Array P2 – Destination Image Array P3 – Size in pixels of the template patch that is used to compute weights. P4 – Size in pixels of the window that is used to compute a weighted average for the given pixel. P5 – Parameter regulatingfilter strength for luminance component. P6 – Same as above but for color components // Not used in a grayscale image.

Implementation:

```
 # importing libraries
import numpy as np
import cv2
from matplotlib import pyplot as plt

img_times = cv2.fastNlMeansDenoising(img_times,None,30,7,21)
   #denoising
```

```

```

3.5.2 Masking

Image masking is the process of separating an image from its background, either to cause the image to stand out on its own or to place the image over another background.Masking is setting certain pixels of an image to some null value such as zero. The reason this is done is that for some reason parts of an image are not to be examined. Here we use masking to select black pixels and non-black pixels in a numpy array.

Implementation:

```

    image_copy = img_times.copy()

     black_pixels_mask = np.all(img_times == [0, 0, 0], axis=-1)
     #selecting black pixels

     non_black_pixels_mask = np.any(img_times != [0, 0, 0], axis
=-1)
# or non_black_pixels_mask = ~black_pixels_mask
#selecting non-black pixels

     image_copy[black_pixels_mask] = [255, 255, 255]
     image_copy[non_black_pixels_mask] = [0, 0, 0]

     #image_copy1 = cv2.blur(image_copy, (5, 5))
     image_copy1 = cv2.fastNlMeansDenoising(image_copy,10,30,7,21)

     filename='blackselect.jpg'
     filename1='blackselectdnois.jpg'

     cv2.imwrite(filename,image_copy)
     cv2.imwrite(filename1,image_copy1)


     print("Non black images")
     plt.imshow(image_copy)
     plt.show()



```

3.5.3 Generalising black

Here the black text in the image inputted to the model were not pure black. Same was the case of the white background. So we generalized a range of pixel values as black. That is, we considered the range of numpy vaues [0,0,0] to [180,255,40] as black.

Implementation:

```
hsv = cv2.cvtColor(img_times, cv2.COLOR_BGR2HSV)
lower_black = np.array([0,0,0])
upper_black = np.array([180,255,40])
mask = cv2.inRange(hsv, lower_black, upper_black)
res = cv2.bitwise_and(img_times, img_times, mask= mask)
print("imgtimes")
plt.imshow(img_times)
plt.show()

print("mask")
plt.imshow(mask)
plt.show()
filemask='masking.jpg'
cv2.imwrite(filemask,mask)

print("res")
plt.imshow(res)
plt.show()


```

3.5.4 Smoothening image

Here we use Image flitering to remove noise and smoothen the image.images also can be filtered with various low-pass filters (LPF), high-pass filters (HPF), etc. A LPF helps in removing noise, or blurring the image.So we use LPF to sommthen the image.

OpenCV provides a function, cv2.filter2D(), to convolve a kernel with an image. As an example, we will try an averaging filter on an image. A 5x5 averaging filter kernel can be defined as follows:

$$K = \frac{1}{25}\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

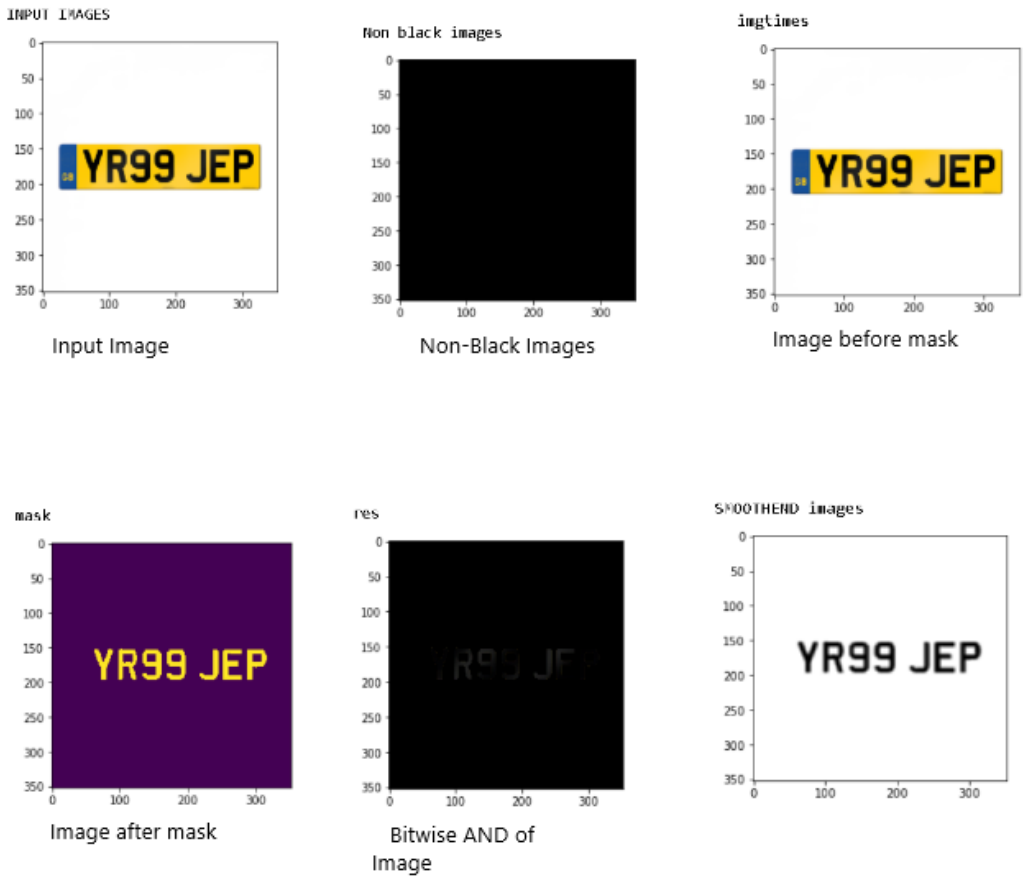

Figure 10: Image Preprocessing

Filtering with the above kernel results in the following being performed: for each pixel, a 5x5 window is centered on this pixel, all pixels falling within this window are summed up, and the result is then divided by 25. This equates to computing the average of the pixel values inside that window. This operation is performed for all the pixels in the image to produce the output filtered image.

Implementation:

```
img_times= cv2.bitwise_not(mask)
img_times=cv2.cvtColor(img_times, cv2.COLOR_BGR2RGB)

kernel= np.ones((5,5), np.float32)/25
img_times = cv2.filter2D(img_times, -1, kernel)


print("SMOOTHEND images")
plt.imshow(img_times)
plt.show()
```

3.6 Segmentation

Once the pre-processing produces a clean character image, it's then segmented into several subcomponents. Segmentation is the process of breaking the whole image into subparts to further process them. Segmentation of image is done in the following sequence :

- Line level Segmentation

- Character level Segmentation

3.6.1 Line level segmentation

Line segmentation is the first and a primilary step for text based image segmentation.It includes horizontal scanning of the image, pixel-row by pixel-row from left to right and top to bottom. At each pixel of each row, the value is taken and stored as unique values in another list . If the unique values in the list is greater than 1, we select the pixels till the next time we get the unique values in the list greater than 1.

Implementation:

```
        start_i=[]
         prev_i=1
```

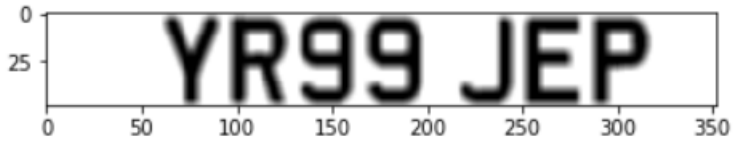

Figure 11: Line-level Segmentation

```
4
5
6               for i in range(img_times.shape[0]):
7                 u = np.unique(img_times[i])
8                 if(len(u)>1):
9                   if(prev_i==1):
10                    start_i.append([i,i+1])
11                    prev_i=0
12                  start_i[-1][1]=i
13                else:
14                  if(len(u)==1):
15                    prev_i=1
16
17              print(start_i)
18
19              cntr=0
20
21              sentenses=[]
22              for i in start_i:
23                cropped = img_times[i[0]:i[1],:img_times.shape[1]]
24                io.imsave("temp/"+name+"_"+str(i[0])+".png",cropped
    )
25                sentenses.append(cropped)
26                plt.imshow(cropped)
27                plt.show()
28                print("cropped", cropped)
29
30
31              start_j=[]
32              prev_i=1
33              cntr=-1
34
35
36              for lines in sentenses:
37                print("Sentenses")
38                plt.imshow(lines)
39                plt.show()
40                print(sentenses)
41                linecntr+=1
42                whitecntr=0
43                cntr+=1
44                out_arr = np.asarray(lines)
45
46
47
```

Here each line of the text is cropped and stored in a folder temp.

3.6.2 Character level segmentation

Character segmentation is the next level of segmentation. It includes vertical scanning of the image, pixel-row by pixel-row from left to right and top to bottom. At each pixel of each row, the value is taken and stored as unique values in another list . If the unique values in the list is greater than 1, we select the pixels till the next time we get the unique values in the list greater than 1.

Implementation:

```
1
2          for j in range(lines.shape[1]):
3       slices=out_arr[:,j]
4       u = np.unique(slices)
5       #print(u)
6       if(len(u)>1):
7         if(prev_i==1):
8           start_j.append([cntr,j,j+1])
9           prev_i=0
10
11        if(start_j[-1][2]<j):
12          start_j[-1][2]=j
13      else:
14        if(len(u)==1):
15          prev_i=1
16
17
18         print("start_j",start_j)
19          i=0;
20
21          i=0
22          prevline=0
23
24          for chrtr in start_j:
25            print("previous line 1===",prevline)
26            if(prevline!=chrtr[0]):
27              i=0
28              prevline=chrtr[0]
29              print("previous line ",prevline)
30            cropped=sentenses[chrtr[0]][:,chrtr[1]-2:chrtr[2]+2]
31            print("newPlot")
32            plt.imshow(cropped)
33            plt.show()
34            lettercntr+=1
35            if(istest==0):
36              io.imsave("dataset/"+name+"/"+str(chrtr[0])+"_"+str(i
    ).zfill(2)+".png",cropped)
37            else:
38              io.imsave("testset_chars/"+name+"_"+str(chrtr[0])+"_"
    +str(i).zfill(2)+".png",cropped)
39            #sentenses.append(cropped)
```

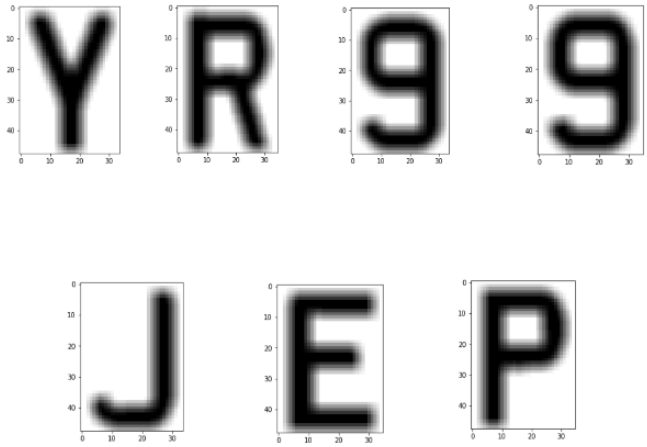

Figure 12: Character-level Segmentation

```
40                i+=1
41            return (linecntr,lettercntr)
42
43
```

Here each character of the text is segmented and stored in the folder testset-chars.

3.7 Character recognition

Finally, the segmented characters are inputted to the OCR Engine, which predicts the chracters in the image. Here, we make a Tensorflow model for OCR which makes use of the Convolutional Neural Networks (CNN) classifier model.

3.7.1 Implementation of the proposed Model

1. *Setup*

In this step we need to import Keras and other packages that we're going to use in building the CNN. We Import the following packages:

- Sequential is used to initialize the neural network.
- Convolution2D is used to make the convolutional network that deals with the images.
- MaxPooling2D layer is used to add the pooling layers.
- Flatten is the function that converts the pooled feature map to a single column that is passed to the fully connected layer.
- Dense adds the fully connected layer to the neural network.

```
1
2  import shutil
3  from PIL import Image
4  import numpy as np
5
6  import tensorflow as tf
7
8  from tensorflow.keras.models import Sequential
9  from tensorflow.keras.layers import Conv2D
10 from tensorflow.keras.layers import MaxPooling2D
11 from tensorflow.keras.layers import Flatten
12 from tensorflow.keras.layers import Dense
13 from tensorflow.keras.layers import Dense, Dropout, Flatten
```

2. *Initializing the Neural Network*

To initialize the neural network we create an object of the Sequential class.

```
1
2      classifier = Sequential()
3
4
```

3. *Convolution*

To add the convolution layer, we call the add function with the classifier object and pass in Convolution2D with parameters. The first argument nb_filter. nbfilter is the number of feature detectors that we want to create. The second and third parameters are dimensions of the feature detector matrix. It's common practice to start with 32 feature detectors for CNNs. The next parameter is input_shape which is the shape of the input image. The images will be converted into this shape during preprocessing. If the image is black and white it will be converted into a 2D array and if the image is colored it will be converted into a 3D array. n this case, we are using colored images. Input_shape is passed in a tuple with the number of channels, which is 3 for a colored image, and the dimensions of the 2D array in each channel. The final parameter is the activation function. Classifying images is a nonlinear problem. So we use the rectifier function to ensure that we don't have negative pixel values during computation. That's how we achieve non-linearity.

```
    classifier.add(Conv2D(64, (3, 3), input_shape = (28, 28, 1),
    activation = 'relu'))

```

4. *Pooling*

In this step we reduce the size of the feature map. Generally we create a pool size of 2x2 for max pooling. This enables us to reduce the size of the feature map while not losing important image information.

```
classifier.add(MaxPooling2D(pool_size = (2, 2)))

```

5. *Flattening*

In this step, all the pooled feature maps are taken and put into a single vector. The Flatten function flattens all the feature maps into a single column.

```
    classifier.add(Flatten())

```

6. *Full connection*

The next step is to use the vector we obtained above as the input for the neural network by using the Dense function in Keras. The first parameter is output_dim which is the number of nodes in the hidden layer. You can determine the most appropriate number through experimentation. The higher the number of dimensions the more computing resources you will need to fit the model. A common practice is to pick the number of nodes in powers of two. The second parameter is the activation function. We usually use the ReLu activation function in the hidden layer.

```
    classifier.add(Dense(units = 128, activation = 'relu'))

```

The next layer we have to add is the output layer. In this case, we'll use the sigmoid activation function since we expect a binary outcome.

```
   classifier.add(Dense(units = 36, activation = 'sigmoid'))

```

7. *Compiling CNN*

We then compile the CNN using the compile function. This function expects three parameters: the optimizer, the loss function, and the metrics of performance.The optimizer is the gradient descent algorithm we are going to use. We use the binary_crossentropy loss function since we are doing a binary classification.

```
    classifier.compile(loss='categorical_crossentropy',optimizer='
    Adam',metrics=['accuracy'])

```

8. *Fitting the CNN*

We are going to preprocess the images using Keras to prevent overfitting. This processing is known as image augmentation. The Keras utility we use for this purpose is ImageDataGenerator.

```

    from keras.preprocessing.image import ImageDataGenerator


```

This function works by flipping, rescaling, zooming, and shearing the images. The first argument rescale ensures the images are rescaled to have pixel values between zero and one. horizontal_flip=True means that the images will be flipped horizontally. All these actions are part of the image augmentation.

```

    train_datagen = ImageDataGenerator(
        rescale=1./255,
        shear_range=0.2,
        zoom_range=0.2,
        horizontal_flip=True)

```

We then use the ImageDataGenerator function to rescale the pixels of the test set so that they are between zero and one. Since this is the test data and not the training data we don't have to take image augmentation steps.

```

    test_datagen = ImageDataGenerator(rescale=1./255)


```

The next thing we need to do is create the training set. We do this by using train_datagen that we just created above and the flow_from_directory function. The flow_from_directory function enables us to retrieve the images of our training set from the current working directory. The first parameter is the path to the training set.

The second parameter is the target_size, which is the size of the image that the CNN should expect. We have already specified this above as 28x28, so we shall use the same for this parameter. The batch_size is the number of images that will go through the network before the weights are updated. The class_mode parameter indicates whether the classification is binary or not.

```
    train_generator = train_datagen.flow_from_directory(
```

```
        'D:\\MiniProject\\ex\\DataSet\\decimals',
        target_size=(28, 28),
        batch_size=32,
    color_mode='grayscale',
        class_mode='categorical')

```

Now we will create the test set with similar parameters as above.

```
    validation_generator = test_datagen.flow_from_directory(
        'D:\\MiniProject\\ex\\testset',
        target_size=(28, 28),
        batch_size=32,
    color_mode='grayscale',
        class_mode='categorical')

```

Finally, we need to fit the model to the training dataset and test its performance with the test set. We achieve this by calling the fit function on the classifier object.

```
    history=classifier.fit(train_Data, train_Label, epochs=5)

```

9. *Prediction*

Now that the model is fitted, we can use the predict method to make predictions using new images. In order to do this we need to preprocess our images before we pass them to the predict method. To achieve this we'll use some functions from numpy. We also need to import the image module from Keras to allow us to load in the new images.

```
    import numpy as np
    from tensorflow.keras.preprocessing import image

```

The next step is to load the image that we would like to predict. To accomplish this we use the load_img function from the image module. The first argument this function takes is the path to the location of the image and the second argument is the size of the image. The size of the image should be the same as the size used during the training process.

```
    for file in os.listdir(test_path):
    # read the image
    file=test_path+file
    img1 = image.load_img(file)
    fileparts=file.split('_')

    img = image.load_img(file, target_size=(28,28,1), grayscale=
    True)


```

we're using 3 channels because our images are color and therefore need to transform this image into a 3D array. To do this we use the img_to_array function from the imagemodule.

```
    import numpy as np
    test_image = image.img_to_array(img)

```

We now have an image with three dimensions. However we are not yet ready to make the predictions because the predict method expects four dimensions. The fourth dimension corresponds to the batch size. This is because in neural networks the data to be predicted is usually passed in as a batch. In this case we have one batch of one input image. We use the expand_dims method from numpy to add this new dimension. It takes the first parameter as the test image we are expanding, and the second parameter is the position of the dimension we are adding.The predict method expects this new dimension in the first position, which corresponds to axis 0.

```
    test_image = np.expand_dims(test_image, axis = 0)

```

Now we use the predict method to predict which class the image belongs to.

```
    pred = classifier.predict(test_image)

```

After running this function we'll get the result: either one or zero. However we don't know which value represents which class. To find out, we use the class_indices attribute of the training set.

```
    predicted_class_indices=np.argmax(pred,axis=1)
    labels = (train_generator.class_indices)

```

Now we plot to compare the inputted image characters and the predicted characters.

```
    labels = dict((v,k) for k,v in labels.items())
    predictions = [labels[k] for k in predicted_class_indices]

    img1=cv2.putText(np.array(img1), predictions[0], (10,30), cv2.
    FONT_HERSHEY_PLAIN,  2,(0,0,255),2,cv2.LINE_AA)
    plt.imshow(img1)
    plt.show()
    if(len(fileparts)>=2):
      if(lcnt!=fileparts[-2]):
        print(txtout)
        txtout=""
        lcnt=fileparts[-2]
    txtout=txtout+" "+predictions[0]

```

Figure 13: Input image for Prediction

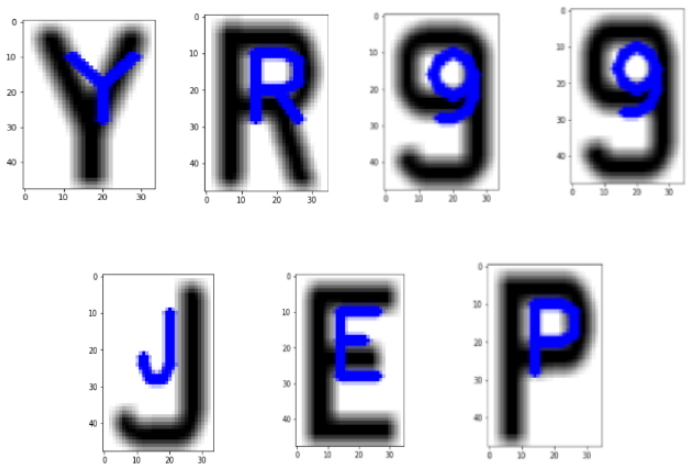

Figure 14: Predicted Alphabets from input image

```
14 print(txtout)
15
```

3.8 Website

After developing the back-end of the project, we have implemented this project as a website. We have used Flask framework to implement our project . In this website , we will be able to see the webcam that we use for the detection and recognition of the license plate and also we will be able to see the history of previously detected license plates stored in the database in tabular form . The previously detected LP lists are stored in descending order of date by default and there is a feature to sort the table based on any of the columns. Another feature has been added that helps the users to search for a particular license plate and also the list of license plates around a particular time period in sorted order, if needed.

4 Conclusion

Automatic Number Plate Recognition (ANPR) is a mass surveillance system that captures the image of vehicles and recognizes their license number. We have presented a robust real-time endto-end ALPR system using the state-of-the-art YOLO object detection CNNs. We trained a network for each ALPR stage, except for the character recognition where letters and digits are recognized separately.ALPR systems typically have three stages: License Plate (LP) detection, character segmentation and character recognition.The entire frame and the vehicle coordinates are used as inputs to train the vehicle detection CNN, the vehicle patch (with a margin) and the coordinates of its LP are used to learn the LP detection network.We report only the results obtained with the Fast-YOLO model in the vehicle and LP detection subsections, since it achieved impressive recall and precision rates in both tasks.

1) Vehicle Detection:

The deep learning YOLOv3 target detection algorithm is applied to vehicle target detection in complex scenes. Through the analysis and comparison of experimental results, the accuracy of this method can reach 89.16detection. The method is greatly improved in accuracy or operational efficiency.The network structure adopted by YOLOv3 is called Darknet-53. This structure adopts the full convolution method and replaces the previous version of the direct-connected convolutional neural network with the residual structure. The branch is used to directly connect the input to the deep layer of the network. Direct learning of residuals ensures the integrity of image feature information, simplifies the complexity of training, and improves the overall detection accuracy of the network. In YOLOv3, each grid unit will have three bounding boxes of different scales for one object. The candidate box that has the largest overlapping area with the annotated box will be the final prediction result. Ad-

ditionally, the YOLOv3 network has three output scales, and the three scale branches are eventually merged.

2) LP Detection:

Every vehicle in the validation set was well segmented with its LP completely within the predicted bounding box. Therefore, we use the vehicle patches without any margin to train the LP detection network. As expected, all LPs were correctly detected in both validation and test sets.

3) Character Segmentation:

A margin of 5its characters fully. Therefore, we double this value (i.e., 10segmentation CNN.

4) Character Recognition:

The padding values that yielded the best recognition rates in the validation set were 2 pixels for letters and 1 pixel for digits. In addition, data augmentation with flipped characters only improved letter recognition, hampering digit recognition. We believe that a greater padding and data augmentation improve letter recognition because each class have far fewer training examples, compared to digits.The best results were obtained with 1 pixel of padding and data augmentation, for both letters and digits. We believe that the proposed ALPR system is robust to locate vehicle, LPs and alphanumeric characters from any other country.

5 Future Work

ALPR can be taken advantage of for vehicle owner identification, vehicle model identification traffic control, vehicle speed control and vehicle location tracking. More features can be added to the existing ALPR to be used as multilingual ALPR to identify the language of characters automatically based on the training data. It can provide more benefits like traffic safety enforcement, security- in case of suspicious activity by vehicle, easy to use, immediate information availability- as compared to searching vehicle owner registration details manually and cost effective for any country. For low resolution images some improvement algorithms like super resolution of images could be focused. Most of the ALPR focus on processing one vehicle number plate but in real-time there can be more than one vehicle number plates while the images are being captured. In multiple vehicle number plate images are considered for ALPR while in most of other systems offline images of vehicle, taken from online database such as are given as input to ALPR so the exact results may deviate from the results. To segment multiple vehicle number plates a coarseto-fine strategy could be

helpful.

6 Bibliography

[1] Elihos, Alperen and Balci, Burak and Alkan, Bensu Artan , Yusuf (2019) , " Deep Learning Based Segmentation Free License Plate Recognition Using Roadway Surveillance Camera Images".

[2] Chirag Patel , Dipti Shah, PhD. , Atul Patel, PhD., "Automatic Number Plate Recognition System (ANPR): A Survey ", International Journal of Computer Applications (0975 –8887)Volume 69–No.9, May 2013

[3] Rayson Laroca, Evair Severo , Luiz A. Zanlorensi , Luiz S. Oliveira, Gabriel Resende Gonc¸alves, William Robson Schwartz and David Menotti , " A Robust Real-Time Automatic License Plate Recognition Based on the YOLO Detector ", arXiv:1802.09567v6 [cs.CV] 28 Apr 2018

[4] Amr Badr, Mohamed M. Abdelwahab, Ahmed M. Thabet, and Ahmed M. Abdelsadek , " Automatic Number Plate Recognition System ", Annals of the University of Craiova, Mathematics and Computer Science Series Volume 38(1), 2011, Pages 62–71 ISSN: 1223-6934

[5] Gupta Mehul , Patel Ankita , Dave Namrata , Goradia Rahul , and Saurin Sheth , " Text-Based Image Segmentation Methodology " , 2nd International Conference on Innovations in Automation and Mechatronics Engineering, ICIAME 2014