



## Table of Contents

<b>1. Introduction</b>	<b>3</b>
1.1 Background of Image processing	3
1.2 The relationship between image processing and linear algebra	4
1.3 Algorithm used (Explanation of image filtering, edge localizing, and using OCR recognition)	5
<b>2. Small examples</b>	<b>19</b>
2.1 Addition and Subtraction	19
2.2 Bilateral Filtering	20
2.3 Canny Edge Detection	21
2.4 Transformation of RGB space	23
2.5 Bilateral Filtering and finding edges for localization	24
<b>3. Real-Life Data Simulation (Python)</b>	<b>28</b>
3.1 Automatic number-plate recognition (ANPR)	28
3.2 Implementation with MobileNet Image classification network(Machine Learning)	29
3.3 Analysis of real-life simulation	30
<b>4. Limitations and Future Work</b>	<b>31</b>
4.1 Limitations	31
4.2 Future work	32
<b>5. Conclusion</b>	<b>32</b>
<b>References</b>	<b>33</b>
<b>Appendix</b>	<b>34</b>

# 1. Introduction

## 1.1 Background of Image processing

With the advancement of digital image sensors, traditional photography has been long forgotten. The majority of photography today is done with digital cameras due to the convenience of being able to access photographs instantaneously without having to wait for films to be printed. An application of digital image sensors is with image processing.

Image processing is the process of performing operations on images in order to improve or extract useful insight for analysis. In this case, we want to apply our knowledge of linear algebra to focus on digital image processing in order to extract specific details for further analysis. Matrices can be used to manipulate digital images since they are matrices when broken down into the smallest pieces of information (pixels). Figure 1 shows binary pixel art, in which each box in the corresponding matrix is represented by either a "1" or a "0." A zero implies that the box is empty (it would ordinarily seem translucent or white), but a one indicates that it is full (in this case it is filled with black).

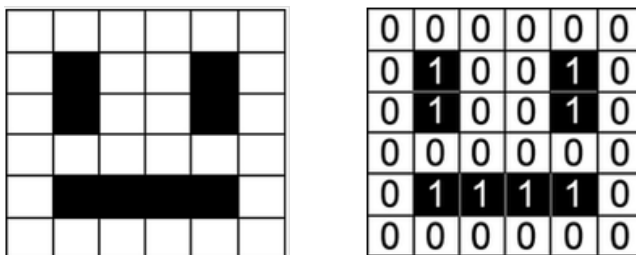


Figure 1

An image is actually a two-dimensional array with a value ranging from 0 (to represent black, the color of the least intensity) to 255 (to represent white, the color of the greatest intensity), which reflects the intensity of the corresponding pixel. A typical digital image has considerably more color complexity than the one displayed in figure 1. As a result, each pixel's value will be a composite of Red, Green, and Blue (RGB) intensity (Figure 2).

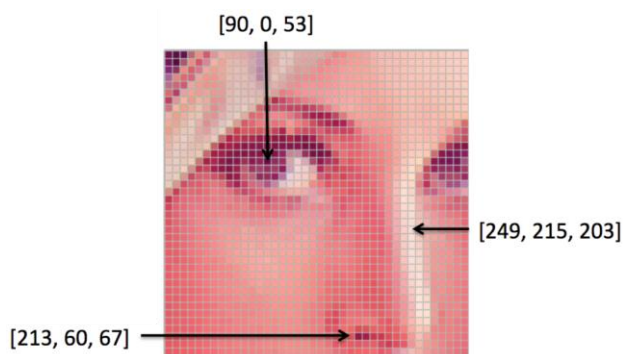


Figure 2

Object detection has received a lot of research attention in recent years because of its close relationship with video analysis and image segmentation. Handcrafted features and shallow trainable architectures are the foundations of traditional object detection methods. By constructing complex ensembles that combine multiple low-level image features with high-level context from object detectors and scene classifiers, their performance quickly plateaus. With the rapid advancement of deep learning, more powerful tools capable of learning semantic, high-level, and deeper features are being introduced to address the issues that exist in traditional architectures. We will provide a conceptual overview of object detection using linear algebra in this report. The report begins with a brief overview of one of the traditional approaches. After which, an example will be provided to demonstrate the mathematical proof of the application. In addition, we will conduct a data simulation test to help relate the use of object detection in real life. There are also experimental analyses provided to compare various methods and draw some meaningful conclusions. Finally, we conclude the report with future research recommendations in both object detection and deep convolutional neural network-based learning systems.

## 1.2 The relationship between image processing and linear algebra

Every digital image is made of pixels and each pixel contains numerical values that can be interpreted as the color in that part of the image. This is where linear algebra comes in as images are represented as a matrix and linear operations. To further illustrate: Consider a gray-scale image & the matrix (Figure 3). Each pixel has a value from 0 to 255. Black is represented by 0 and white is represented by the maximum value 255.

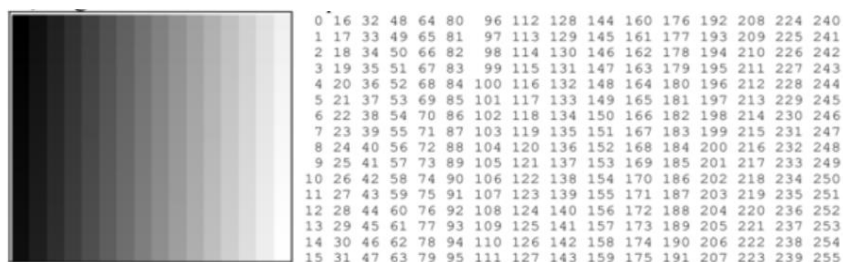


Figure 3

Matrix representation of a color image depends on the color system used by the program that is processing the image. The most popular one is the RGB where each pixel specifies the amount of Red (R) , Green (G) and Blue (B). The pixel is then represented as a tri-dimensional vector (r,g,b). Different combinations of RGB produce different colors (Figure 4).

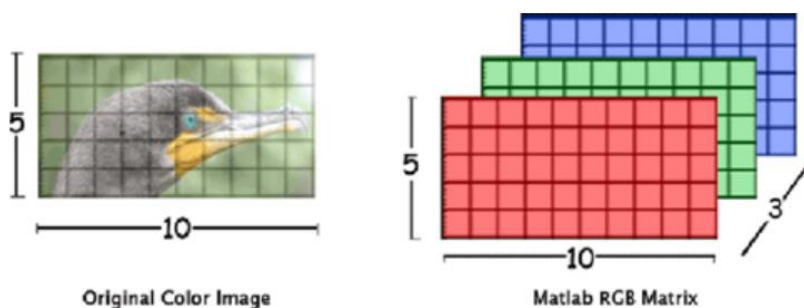


Figure 4

### 1.3 Algorithm used (Explanation of image filtering, edge localizing, and using OCR recognition)

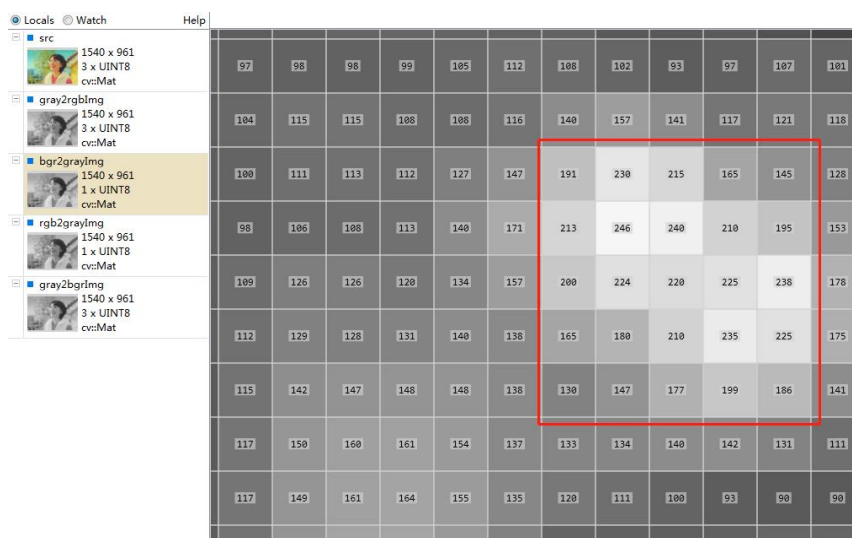
#### Step 1: Converting the image to grayscale

The cv2.imread Application Programming Interface (API) call returns an unit8\_t data type array of the image which is fed into the mathematical equation below.

**RGB[A] to Gray:  $Y = 0.299R + 0.587G + 0.114B$  (OpenCV, Color conversions 3.4.17-dev)**

The resultant Y value is the value of that particular pixel. For example, the lower right corner pixel has the following value

CV\_BGR2GRAY: Lower right corner:  $0.1140 * 163 + 0.5870 * 182 + 0.2989 * 203 \approx 186$



(Kosh et al., 2020)

This calculation is done across all the pixels in the image and the resultant value is assigned to the new grayscale image.


We weigh each channel differently to account for how many colors we perceive in our eyes. We can sense approximately twice as much green as red, because of the cones and receptors in our eyes. Similarly, we detect almost twice as much red as we do blue. As a result, when converting from RGB to grayscale, we make sure to account for this (Rosebrock, 2021). When we don't need color, we typically employ a grayscale representation of a picture (such as in detecting faces or building object classifiers where the color of the object does not matter) (Rosebrock, 2021). We can conserve memory and be more computationally efficient by discarding color. Grayscale images are single-channel images with pixel values in the range  $[0, 255]$  (i.e. 256 unique values).

### Step 2: Applying filter and finding edges for localization

A Bilateral filter is used for smoothening images and reducing noise while **preserving edges**. (Paris et al., 2007)

A Gaussian filter takes the pixels around the neighborhood and finds the Gaussian weighted average. The Gaussian filter does not consider whether a pixel is an edge pixel or not, so it blurs the edges which takes away the important details from the image which we intend to capture. (Paris et al., 2007)

Bilateral filtering is used to blur images and remove noise while preserving the edges. This is accomplished by taking a Gaussian filter in space and considering an additional Gaussian filter which is a function of pixel difference / “minimum” amplitude of an edge. (Khan et al., 2020)

  
THE UNIVERSITY OF AUCKLAND  
NEW ZEALAND

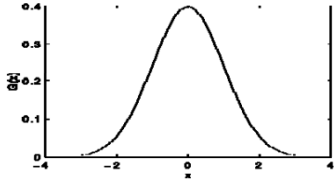
## Gaussian Filtering

Gaussian filtering is used to blur images and remove noise and detail.  
In one dimension, the Gaussian function is:

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}}$$

Where  $\sigma$  is the standard deviation of the distribution. The distribution is assumed to have a mean of 0.

Shown graphically, we see the familiar bell shaped Gaussian distribution.



Gaussian distribution with mean 0 and  $\sigma = 1$

18

Gaussian Filter

$$GB[I]_p = \sum_{q \in S} G_{\sigma}(\|p - q\|) I_q$$

↓  
Normalized Gaussian  
Function



Gaussian function of space

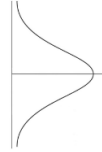
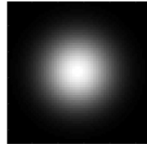
The parameters  $p$  and  $q$  in the above Gaussian filter are the position of pixels.  $GB[I]_p$  is the result at pixel  $p$ , and the on the right-hand side of the equation (RHS) is essentially a sum over all pixels  $q$  weighted by the Gaussian function.  $I_q$  is the intensity at pixel  $q$ . The above Gaussian filter ( $G_{\sigma}$ ) is a spatial Gaussian that decreases the influence of distant pixels.  $\|p - q\|$  is the Euclidean distance between pixel locations  $p$  and  $q$ . (Paris et al., 2007)  $\sigma$  is a parameter defining the extension of the size of the neighborhood. (Khan et al., 2020)

$$BF[I]_p = \frac{1}{W_p} \sum_{q \in S} G_{\sigma_s}(\|p - q\|) G_{\sigma_r}(|I_p - I_q|) I_q$$

↓  
Normalization  
Factor

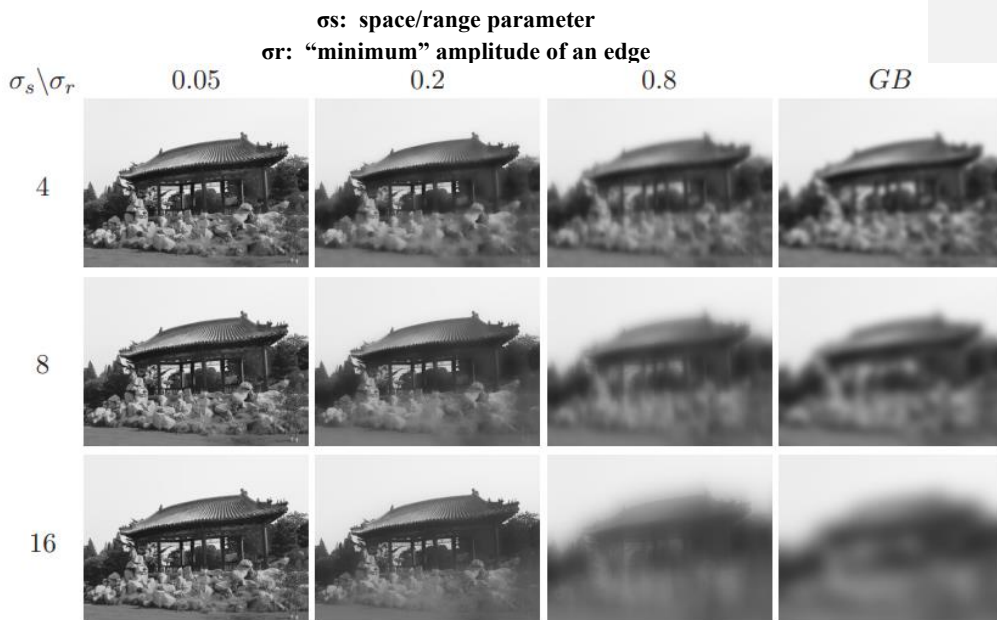
↓  
Space Weight

↓  
Range Weight



The first Gaussian function of space ( $\sigma_s$ ) only considers spatial neighbors, that is pixels that appear close together in the  $(x, y)$ -coordinate space of the image. (Paris et al., 2007) The second Gaussian function of intensity difference ( $\sigma_r$ ) models the pixel intensity of the neighborhood, ensuring that only pixels with similar intensity are included in the actual computation of the blur (Paris et al., 2007).





(Paris, 2007)

An important characteristic of bilateral filtering is that the weights are multiplied, which implies that no smoothing occurs as soon as one of the weights is close to 0. As a result, as long as the range sigma is smaller than the amplitude, raising the spatial sigma has no effect on the edge. For example, regardless of the spatial setting, the curve of the roof remains unchanged for tiny range values. The range values are supplied with the intensities ranging from [0, 1] (Paris, 2007)

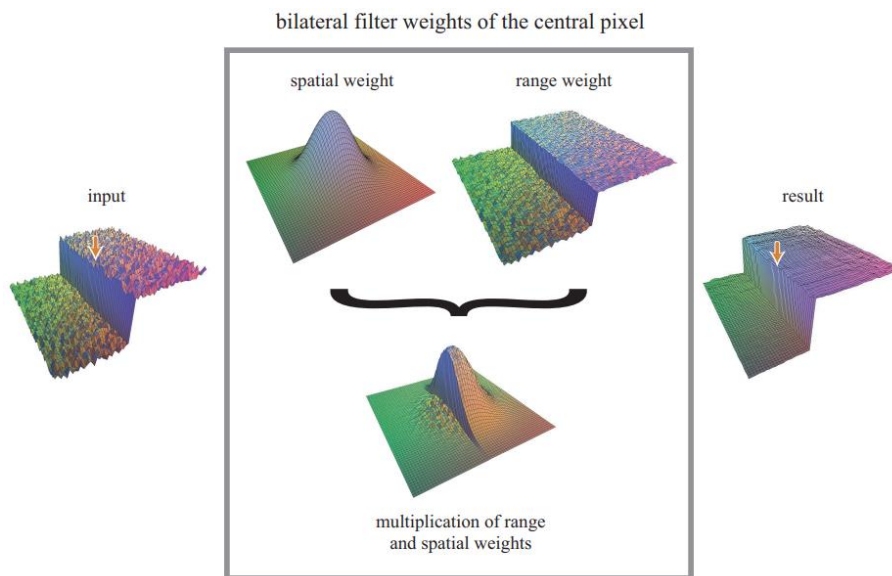
**Parameters:**  $\sigma_s$  and  $\sigma_r$  will measure the amount of filtering for the image I.

- As the range parameter  $\sigma_r$  increases, the bilateral filter becomes closer to Gaussian blur because the range Gaussian is flatter i.e., almost a constant over the intensity interval covered by the image (Paris et al., 2007)
- If the spatial parameter  $\sigma_s$  is increased, a smoothing effect occurs on larger features (Paris et al., 2007)

Therefore, a combination of both components ensures that only nearby similar pixels and similar intensity levels contribute to the final result. No smoothing occurs when the weights are 0 (Paris et al., 2007).

The bilateral filter smooths a picture while maintaining its edges. A weighted average of its neighbors is used to replace each pixel. A spatial component penalizes distant pixels, whereas a range component penalizes pixels with varying intensities. The combination of both components guarantees that only pixels that are close in proximity to one another contribute

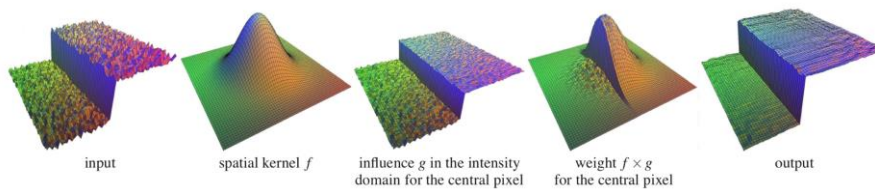
to the final outcome. The weights are represented for the central pixel (under the arrow) (Paris et al., 2007).



(Paris et al., 2007)

The advantage of Bilateral filtering is that it can be computed at interactive speed even on large images thanks to efficient techniques such as downsampled piecewise-linear acceleration, nearest-neighbor downsampling, and Fast Fourier Transform were a  $O(n^2)$  convolution becomes a  $O(n)$  multiplication in the frequency domain. (Paris et al., 2007)

The bilateral filtering process can be summarized as in the image shown below.



(Paris, 2007)

### Step 3: Canny Edge Detection

There are 4 Stages in a multi-stage edge detection algorithm.

#### A. Noise reduction

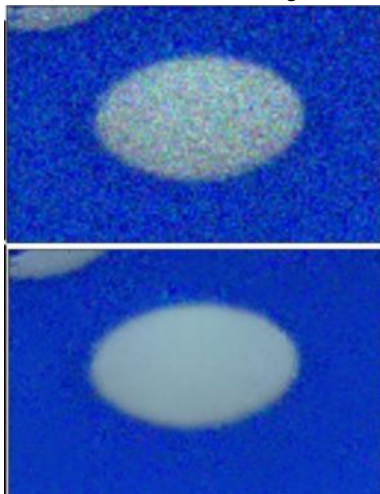
- Using a 5x5 Gaussian filter
  - Using spatial filters (1D) where  $\sigma$  is the standard deviation of the distribution

$$G(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}}$$

- For (2D)

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

- Using the 2D convolution(Image Filtering), it acts as a HPF(High-pass filter), which aids in removing noise and finding edges in images.



Example of using HPF

- To achieve a  $\sigma$  which sets the right filtering rate, Gaussian distribution is required to resize the pixel(matrix) in terms of increments

$$\frac{1}{273}$$

1	4	7	4	1
4	16	26	16	4
7	26	41	26	7
4	16	26	16	4
1	4	7	4	1

Example of a 5x5 pixel with a Discrete approximation to Gaussian function with  $\sigma=1.0$

#### B. Finding Intensity Gradient of the image

- The Smoothed image is then filtered with a Sobel kernel in both horizontal and vertical direction to get the first derivative in the horizontal direction ( $G_x$ ) and vertical direction ( $G_y$ ). From these two images, we can find edge gradient and direction for each pixel as follows:

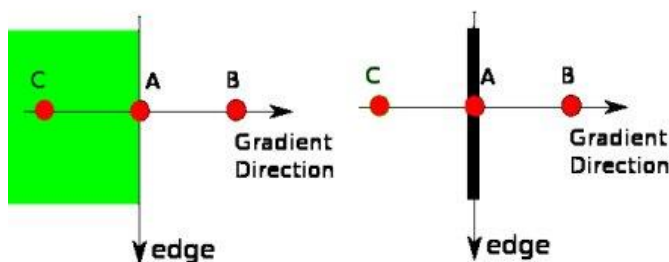
$$Edge\_Gradient (G) = \sqrt{G_x^2 + G_y^2}$$

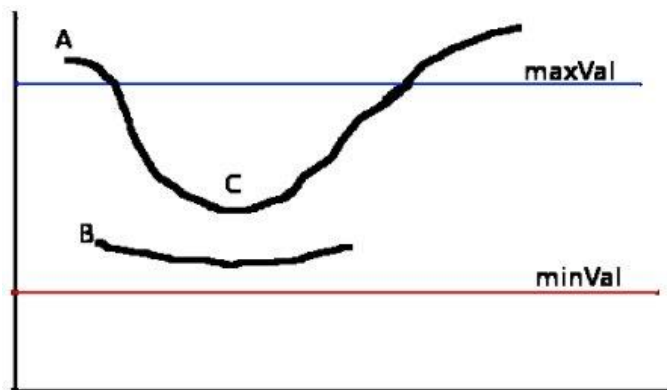
$$Angle (\theta) = \tan^{-1} \left( \frac{G_y}{G_x} \right)$$

The gradient is always perpendicular to edges which are rounded to one of four angles; vertical, horizontal, and two diagonal directions

#### C. Non - maximum Suppression

After achieving the direction and magnitude of the gradient, unwanted pixels which may not constitute the edge are removed where the pixels are checked if it is a local maximum in its range in the direction of the gradient.





Point A is on the edge ( in the vertical direction). The gradient direction is normal to the edge. Points B and C are in gradient directions. So point A is checked with points B and C to see if it forms a local maximum. If so, it is considered for the next stage, otherwise, it is suppressed ( put to zero).

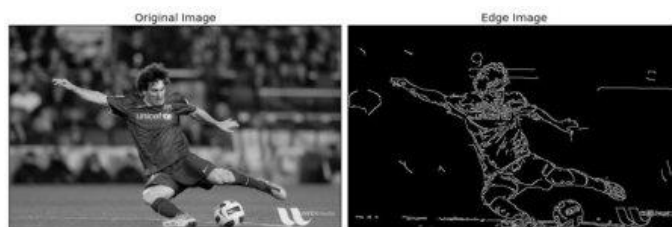
In short, the result you get is a binary image with "thin edges".

#### D. Hysteresis Thresholding

two threshold values, minVal and maxVal to decide the finalized edges Edges are considered if  $Edges > maxVal$  and not considered if  $Edges < minVal$ . The edge will be considered by its connectivity if  $minVal < Edges < maxVal$ . If they are connected to "sure-edge" pixels, they are considered to be part of edges. Otherwise, they are also discarded.

The edge A is above the maxVal, so it is considered as a "sure-edge". Although edge C is below maxVal, it is connected to edge A, so that is also considered a valid edge and we get that full curve. But edge B, although it is above minVal and is in the same region as that of edge C, it is not connected to any "sure-edge", so that is discarded. So it is very important that we have to select minVal and maxVal accordingly to get the correct result.

Hence the finalized outcome would be



#### Step 4: Finding Contours and Applying Mask

The arguments for the `cv2.findContours` API (Application Programming Interface) is the source picture, followed by the contour retrieval mode and the contour approximation technique. It generates the outlines and hierarchy. `contours` is a Python list object of all the image's contours. Each contour is a Numpy array of (x,y) coordinates of the object's border points.

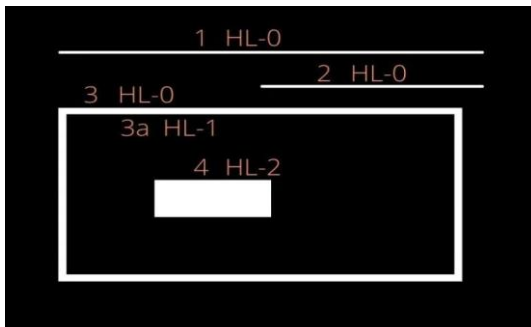
The CHAIN\_APPROX\_SIMPLE method compresses horizontal, vertical, and diagonal segments along a contour, leaving just their ends. This indicates that all points along straight lines will be ignored, leaving just the final points. It removes all redundant points and compresses the contour, thereby saving memory.

#### Contour Hierarchies: Parent-Child relationship

The parent-child relationship between contours is represented by hierarchies.

- All the individual numbers, i.e., 1, 2, 3, and 4 are separate objects, according to the contour hierarchy and parent-child relationship.
- We can say that the 3a is a child of 3. Note that 3a represents the interior portion of contour 3.
- Contours 1, 2, and 4 are all parent shapes, without any associated child, and their numbering is thus arbitrary.

`RETR_TREE` retrieves all the contours. It creates a complete hierarchy, with the levels not restricted to 1 or 2. Each contour can have its own hierarchy, in line with the level it is on, and the corresponding parent-child relationship that it has.



Using the coordinates from the contours NumPy array, we iterate through the array and approximate a polygon with another with fewer vertices so that we can retain the bulk of the information but in a less complex state.

The next few steps involve manipulation of the image and extracting the edge pixels of the number plate as if it were in a bounding box, where `x1`, `x2`, `y1`, `y2` are the vertices of the number plate.

### Step 5: Using OCR To Read Text

Finally, we use a pre-trained model from the easyOCR library to detect the words, extract them as text and place a bounding box over them.

### What is OCR

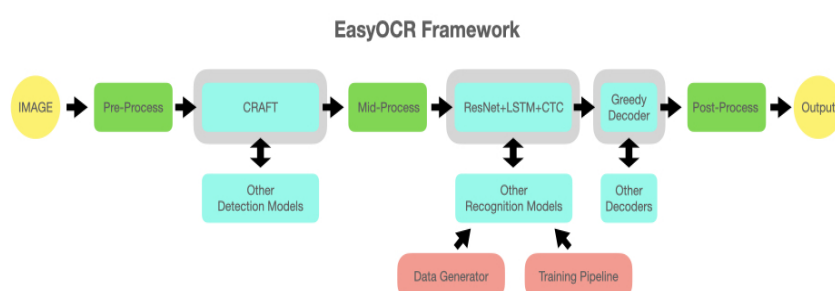
OCR, formerly known as Optical Character Recognition, is a revolutionary technology in the digital world. OCR is actually a complete process under which the images/documents which are present in a digital world are processed and from the text are processed out as normal editable text.

### Purpose of OCR

OCR is a technology that enables you to convert different types of documents, such as scanned paper documents, PDF files, or images captured by a digital camera into editable and searchable data.

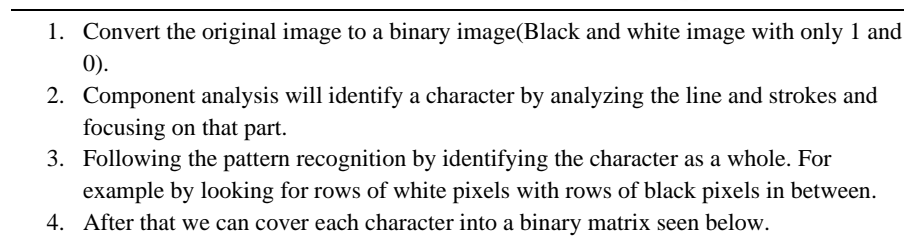
### What is EasyOCR

EasyOCR is a python library that holds PyTorch as a backend handler. It supports 42+ languages for detection purposes.



EasyOCR does certain pre-processing steps(gray scaling and etc.,) within its library and extracts the text. It also applies the CRAFT(Character Region Awareness for Text Detection) algorithm to detect the text. CRAFT is a scene text detection method to effectively detect text areas by exploring each character and affinity between the characters. The recognition model uses CRNN. The sequencing labeling is performed by LSTM and CTC(Connectionist Temporal Classification), here the CTC is meant for labeling the unsegmented sequence data with RNN.

We will be implementing EasyOCR in this study.

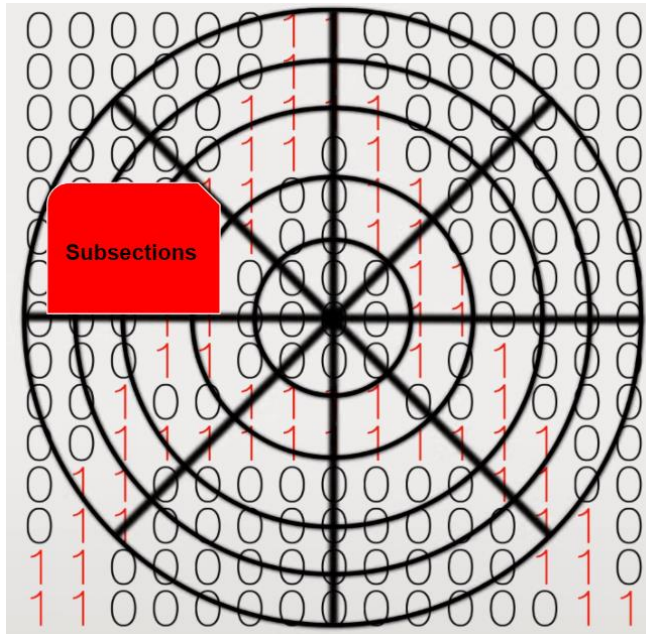




5. Then by using the euclidean distance formula we can find the distance from the center of the matrix to the furthest one.

$$d = \sqrt{(y_2 - y_1)^2 + (x_2 - x_1)^2}$$

6. We then create a circle of that radius and split it up into more granular sections.



7. Then the algorithm will compare every single sub-section and can send a database of matrices representing characters with various fonts to find a character it statistically has the most in common.
8. Doing this for every line and character and then organizing all the characters together
9. In the end, the image is analyzed and converted to text.

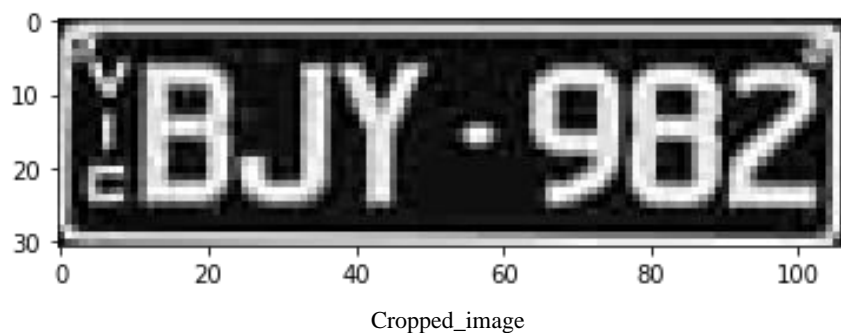
Using easyOCR extracting text from the image

```
reader = easyocr.Reader(['en'])
result = reader.readtext(cropped_image)
result
```

1. Use Reader class from EasyOCR class and then pass ['en'] as an attribute which means that now it will only detect the **English** part of the image as text, if it will find other languages like Chinese and Japanese then it will ignore those text.
2. Here we are loading the cropped-image in the readText() function and we will have the result below.

Out[26]: [[([0, 2], [107, 2], [107, 31], [0, 31]), 'EBJY. 982'], 0.3741937983327902]]

- The first list represents the coordinate of the text inside the cropped-image



- The second list represents the text that easyOCR has recognized.
- The last list represents the accuracy/confidence level. In this case, is 37%

3. After that, the last line just shows the result and the text overlaid with the image.



## 2. Small examples

### 2.1 Addition and Subtraction

To illustrate the workings of the algorithm, two matrices A and B (images) with the same dimension will be used. Adding the two matrices (shown below) will result in new matrices with the same dimension (new image)

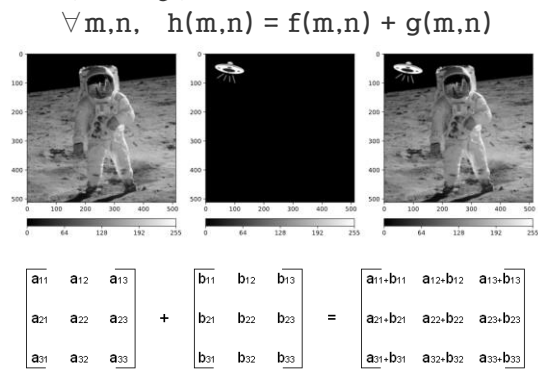


Figure 5: The image on the right is the sum between the two images on the left.

Now, the equation for subtraction is:

$$\forall m,n, \quad h(m,n) = f(m,n) - g(m,n)$$

or  $\forall m,n, \quad h(m,n) = |f(m,n) - g(m,n)|$

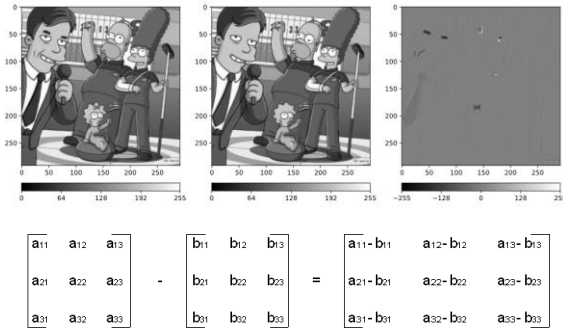


Figure 6: The image on the

Note that the image of difference has values between  $-255$  and  $255$ .

## 2.2 Bilateral Filtering

The below shown Input Image is convolved with a  $G(\sigma_s)$  and  $G(\sigma_r)$



Gaussian Matrix with  $\sigma_s = 1.0$

$$\frac{1}{273} \begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

(Wolfart et al., 2003)

The matrix for computing  $G(\sigma_r)$  for the first pixel computed from CPython.

$$BF[I]_p = \frac{1}{W_p} \sum_{q \in S} G_{\sigma_s}(\|p - q\|) G_{\sigma_r}(I_p - I_q) I_q$$

```
[0.15915494309189535, 0.15915494309189535, 0.15915494309189535,
0.15915494309189535, 0.15915494309189535, 0.15915494309189535,
0.15915494309189535, 0.15915494309189535, 0.15915494309189535,
0.15915494309189535, 0.15915494309189535, 0.15915494309189535,
```

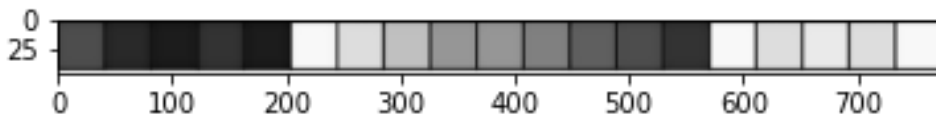
```
0.15915494309189535, 0.15915494309189535, 0.15915494309189535,
0.15915494309189535, 0.15915494309189535, 0.15915494309189535,
0.15915494309189535, 0.15915494309189535, 0.15915494309189535,
0.15915494309189535, 0.15915494309189535, 0.15915494309189535,
0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

The matrix for computing  $G(\sigma_r)$  for the second pixel computed from CPython.

```
[0.0, 0.0, 0.0, 0.15915494309189535, 2.794264393012029e-64,
0.021539279301848634, 0.0, 0.0, 0.0, 0.15915494309189535,
2.794264393012029e-64, 0.021539279301848634, 0.0, 0.0, 0.0,
0.15915494309189535, 2.794264393012029e-64, 0.021539279301848634,
0.0, 0.0, 0.0, 0.15915494309189535, 2.794264393012029e-64,
0.021539279301848634, 2.1425435990548656e-116,
2.1425435990548656e-116, 2.1425435990548656e-116,
2.1425435990548656e-116, 2.1425435990548656e-116,
2.1425435990548656e-116, 0.0, 0.0, 0.0, 4.374827215058053e-44,
2.1425435990548656e-116, 2.2064338800529126e-50]
```

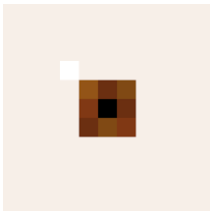
Once all the matrices have been computed, they are multiplied with `im[p_y, p_x]`, to get the final value of the pixel at that particular location.

The output image from the bilateral filtering is

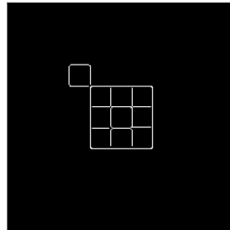


## 2.3 Canny Edge Detection

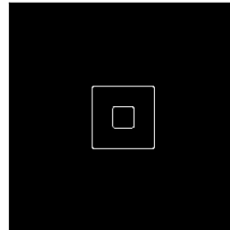
Original image



No-Edged Image



Edged Image



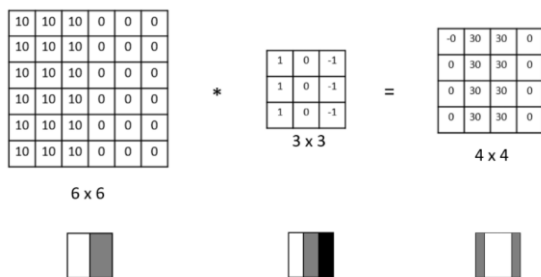
Using the equation below, the threshold values  $i(\text{minVal})$  and  $j(\text{maxVal})$  set the boundaries for which values in the matrix to pass. This threshold is used to form a kernel matrix which is used to filter;  $\text{minVal} < \text{Edges} < \text{maxVal}$ .

$$H_{ij} = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{(i - (k+1))^2 + (j - (k+1))^2}{2\sigma^2}\right); 1 \leq i, j \leq (2k+1)$$

```
edges = cv.Canny(img,0,0)
edges2 = cv.Canny(img,400,500)
```

```
#no threshold
plt.subplot(121),plt.imshow(edges,cmap = 'gray')
plt.title('No-Edged Image'), plt.xticks([], plt.yticks([]))
```

```
#higher threshold
plt.subplot(122),plt.imshow(edges2,cmap = 'gray')
plt.title('Edged Image'), plt.xticks([], plt.yticks([]))
plt.show()
```



This image shows how the original image is multiplied with a filter matrix, set using the threshold combination, to form two distinct parts that would portray the distinct line in the edged image.

## 2. Demonstration of the algorithm through example

To illustrate the various algorithms described in the previous section, simple code examples are demonstrated here to showcase the inner workings of some of the steps used for ANPR detection.

The general outline is shown as in the below image. The image is first converted to grayscale for easier manipulation and ease of processing. Additionally, bilateral filtering is applied to the previous result so as to filter on the 'internal' pixels of an object but leave the edges alone. Then, convolution is applied to the image using the Canny edge detection algorithm using the Sobel kernels and 5x5 Gaussian filter. Finally, contour detection and masking is done so as to retrieve just the coordinates of the number plate and use OCR to extract in a text.

### 2.4 Transformation of RGB space

The first step is reading the below image and converting it into `unit8_t` array data type.



Number of rows = 900

Number of Columns = 1200

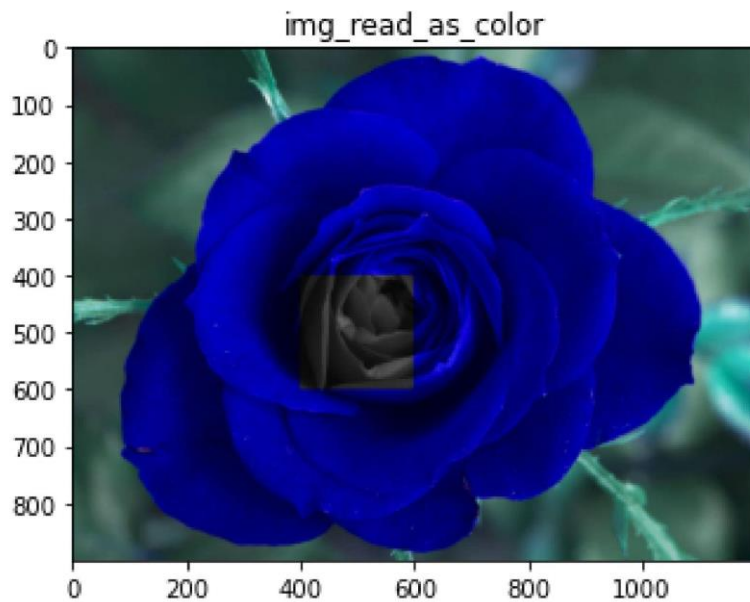
The channel order is retained when passing in the RGB values. If one passes the wrong order of channels, the converted output will differ.

Using a simple transformation,

```
for row in range(400,600):
    for column in range(400,600):
        image1[row][column] = image1[row][column][0] * 0.114 + image1[row][column][1] * 0.587 +
image1[row][column][2] * 0.299

plt.imshow(image1)
```

```
plt.title('img_read_as_color')
plt.show()
```



Note: The “Weird blue and green color” is the viridis colormap that pyplot uses by default, thus matplotlib by inheritance.

As shown in the above image, a rectangular portion of the image is in grayscale while the rest of the image is in viridis colormap, thus this demonstrates the color to grayscale process.

## 2.5 Bilateral Filtering and finding edges for localization

The bilateral filter is a Gaussian convolution that has a large effect on regions of uniform color and a weak effect on regions with a lot of color variation. The bilateral filter functions as an edge-preserving or edge-aware filter since we expect substantial color variation at edges.

It would be difficult to explain bilateral filtering using pixel values, thus we have written a basic bilateral filtering python code to further solidify our explanation.



```

# Gaussian convolution applied at each pixel in an image
# Parameters:
#   im: Image to filter, provided as a numpy array
#   sigma: The variance of the Gaussian (e.g., 1)

@lru_cache(maxsize=None)
def gaussian(path, sigma):
    im = cv2.imread(path, cv2.IMREAD_COLOR)
    im = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY) # COLOR_BGR2BGR COLOR_BGR2GRAY
    height, width = im.shape
    img_filtered = np.zeros([height, width, 3])

    # Define filter size.
    # A Gaussian has infinite support, but most of its mass lies within three
    standard deviations of the mean.
    # The standard deviation is the square of the variance, sigma.
    n = np.int(np.sqrt(sigma) * 3)

    # Iterate over pixel locations p
    for p_y in range(height):
        for p_x in range(width):
            gp = 0
            w = 0

            # Iterate over kernel locations to define pixel q
            for i in range(-n, n):
                for j in range(-n, n):
                    # Make sure no index goes out of bounds of the image
                    q_y = np.max([0, np.min([height - 1, p_y + i])])
                    q_x = np.max([0, np.min([width - 1, p_x + j])])

                    constant = 1 / (2 * np.pi * sigma ** 2)

                    # Compute Gaussian filter weight at this filter pixel
                    g = constant * np.exp(-((q_x - p_x)**2 + (q_y - p_y)** 2)
                    / (2 * sigma**2))

                    gr_x = abs(im[p_y, p_x] - im[q_y, q_x])
                    gr = constant * np.exp(-(gr_x ** 2) / (2 * sigma**2))

                    # Accumulate filtered output
                    gp += g * im[p_y, p_x] * gr

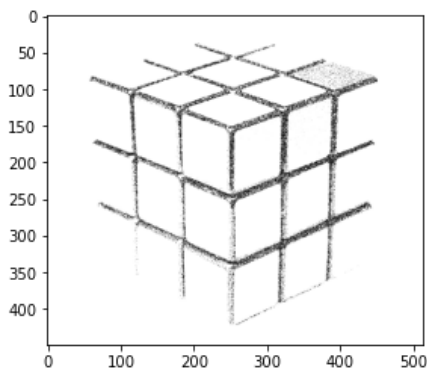
```

```

        # Accumulate filter weight for later normalization, to
maintain image brightness
        w += g
        img_filtered[p_y, p_x, :] = gp / (w + np.finfo(np.float32).eps) #
eps = epsilon
    return img_filtered

img_filtered = gaussian(path="img\\rubiks_cube.png", sigma=1)
plt.imshow(img_filtered)

```



The above image is the result of the standard python implementation of Bilateral Filtering, the run time is 5m 21.7s, this could be improved by using Cython or PyPy or even in plain old C99.

$$\mathcal{N}(d) = e^{-\left(\frac{(q_x - p_x)^2 + (q_y - p_y)^2}{2\sigma^2}\right)}$$

Unnormalized Gaussian (Khan et al., 2020)

Euclidean distance  $|p - q|$  is calculated between the pixels  $p$  and  $q$  and the terms are summed

up and multiplied by the pixel intensity of pixel q. W is the normalization factor to preserve image brightness during filtering (Khan et al., 2020)

$$G_{\sigma}(x) = \frac{1}{2\pi \sigma^2} \exp \left( -\frac{x^2}{2\sigma^2} \right).$$

A constant of  $\frac{I}{2 * \pi * \sigma^2}$  is added (Paris et al., 2007)

$$G(p) = \frac{1}{w} \sum_{q \in S} \mathcal{N}(|p - q|) I_q$$

**Equation I**

Gaussian filtering with normalization via w (Khan et al., 2020)

$$G(p) = \frac{1}{w} \sum_{q \in S} \mathcal{N}(|p - q|) (|I_p - I_q|) I_q$$

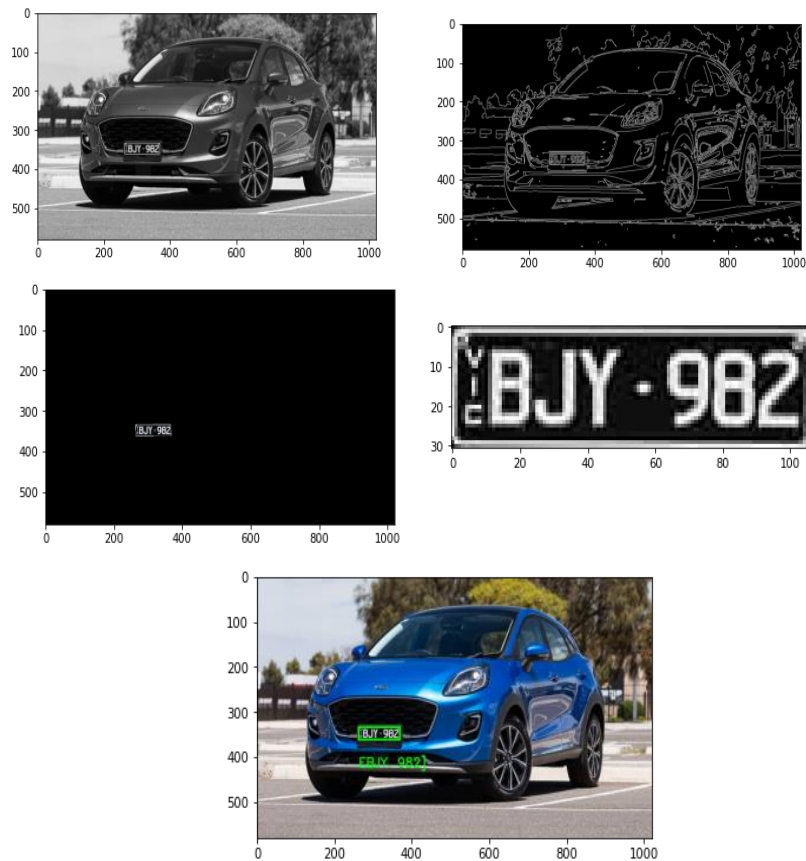
**Equation J**

An additional term is considered in the gaussian filter to preserve detail around the edges (Khan et al., 2020). It turns out that one controllable way to modulate filtering in edge regions is to weight the contribution of each  $I_q$  to  $G(p)$  in **Equation I** by the color difference between p and q (Khan et al., 2020)

### 3. Real-Life Data Simulation (Python)

#### 3.1 Automatic number-plate recognition (ANPR)

Automatic number-plate recognition is a highly accurate system capable of reading vehicle number plates without human intervention. It is a technology that uses optical character recognition (OCR) on images. It first uses a series of image manipulation techniques to detect, normalize and enhance the image of the number plate, and then optical character recognition (OCR) to extract the alphanumerics of the license plate. Below pictures the steps of automatically recognizing a number plate.



### 3.2 Implementation with MobileNet Image classification network(Machine Learning)

Object detection alone has its limitations. To improve its accuracy in detecting images and recognizing text, we would have to apply machine learning.

We will be implementing the MobileNet image classification with Tensorflow on Google Colab for our demo. MobileNet is a class of CNN that was open-sourced by Google, and therefore, this gives us an excellent starting point for training our classifiers that are insanely small and insanely fast. The MobileNet model is designed to be used in mobile applications, and it is TensorFlow's first mobile computer vision model. MobileNet uses depth-wise separable convolutions. It significantly reduces the number of parameters when compared to the network with regular convolutions with the same depth in the nets. The result is a lightweight deep neural network.

Our training data set consisted of 410 labeled images in PNG file format. We simulated our data with steps of 1000, 5000, and 10,000.

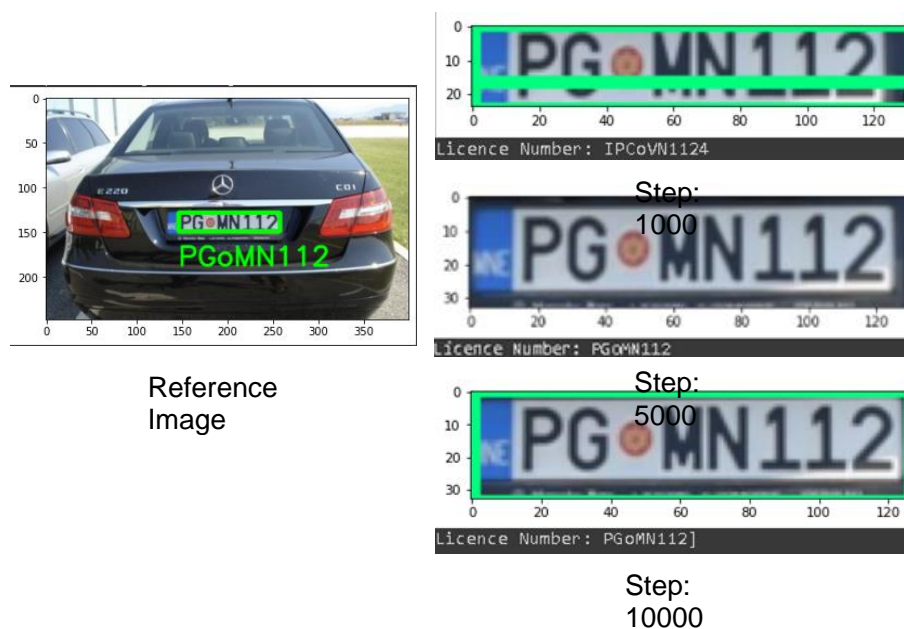
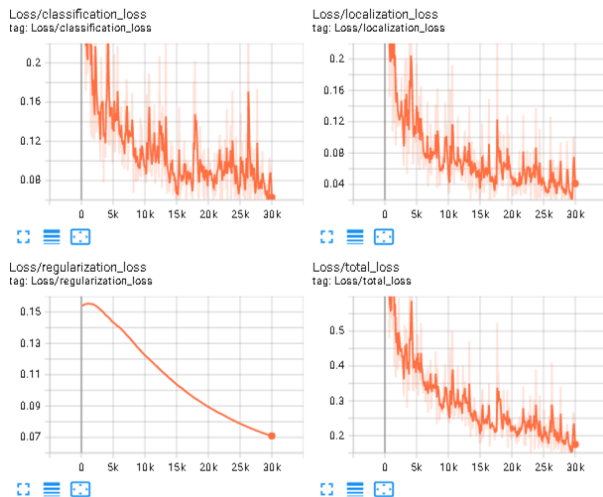


Fig.  
Training  
set

### 3.3 Analysis of real-life simulation



**Fig. Loss metrics**

Increasing the total number of training steps (number of epochs x Number of training steps per epoch) gives us more confidence in extracting the text from the number plate.

The example above also highlights a limitation of image detection - the inability to differentiate between the number 0, the letter O, and a circular shape.

For our study, we realized that most number plates had letters that were uppercase. We noted this observation and wrote a few lines of code to help us omit any lowercase letters that may

have been extracted which gives us greater confidence in extracting the correct plate number from an image.

In totality, we will have to consider the time and cost of training a model when dealing with sizable data.

## 4. Limitations and Future Work

### 4.1 Limitations

- Internet connectivity (For cloud-based systems)
  - If the bandwidth does not allow for a continuous video stream of the monitored areas
- Camera quality (due to extreme weather or lighting)
  - Hindrances like heavy rain or snow can affect the accuracy of license plate recognition
- Privacy (Hackers)
  - The system is mostly cloud based, which allows authorities to attain the full information of records of a particular car owner just by scanning the license plate
  - It is a potential threat to many users as their privacy, eg. home addresses or billing information, can be easily exposed by a hacker who can bypass the camera security
- Huge power requirements for training the deep learning model
  - As the system is accessing an extremely huge amount of data within microseconds, the computing power consumes around 300W
- Misread characters (mild issue)
  - 'O' and '0' are commonly mixed up

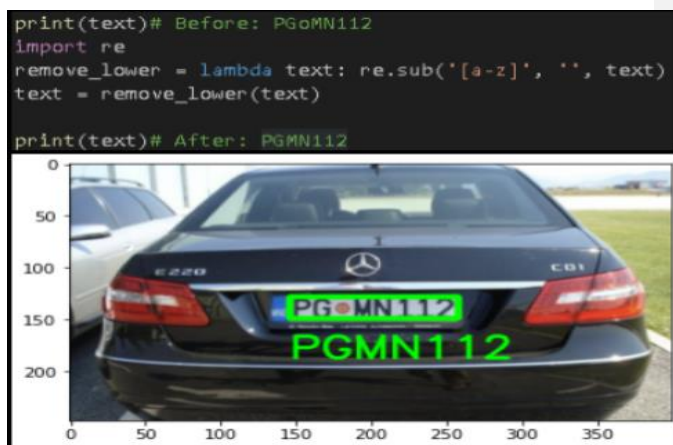


Fig. Removal of Lowercase characters



Image displaying the potential error of distinguishing similar letters and

numbers

- Does not account for human error (grace period)
  - Drivers who enter the carpark and are unavailable in finding a parking space
  - ANPR camera catches them enter and leave which leads to a fine

## 4.2 Future work

- Higher resolution digital cameras
- LED illuminations
- Use FPGAs to accelerate deep learning performance by designing custom hardware architectures solving latency issues
- Faster and lower power consuming general-purpose processors
- Increase network bandwidth using 5G mobile networks
- More robust algorithm to adapt to non-standardized formats, irrespective of regions
  - Matrix combinations that allow to work in low and extremely high lighting areas and filter through glares
- All proposed/designed algorithms need to be tested for real-time scenarios rather pre-acquired images

## 5. Conclusion

Given the use of ANPR in everyday life, it is a helpful technical tool that may aid authorities in managing traffic and preventing crimes. Edge detection devices such as Google's Tensor processing unit and Nvidia Jetson Nano hugely assist in deploying an ANPR system.



## References

References for reading in image as grayscale and bilateral filtering according to IEEE style

1. Kosh, vasilykarasev, Atif Anwer, and AlfredBr, “Why would cv2.color\_rgb2gray and cv2.color\_bgr2gray give different results?,” *Stack Overflow*, 11-Jul-2020. [Online]. Available: <https://stackoverflow.com/questions/62855718/why-would-cv2-color-rgb2gray-and-cv2-color-bgr2gray-give-different-results>. [Accessed: 26-Feb-2022].
2. “Color conversions,” *OpenCV*. [Online]. Available: [https://docs.opencv.org/3.4/de/d25/imgproc\\_color\\_conversions.html](https://docs.opencv.org/3.4/de/d25/imgproc_color_conversions.html). [Accessed: 26-Feb-2022].
3. A. Rosebrock, “OpenCV color spaces ( cv2.cvtColor ),” *PyImageSearch*, 09-May-2021. [Online]. Available: <https://pyimagesearch.com/2021/04/28/opencv-color-spaces-cv2-cvtColor/>. [Accessed: 26-Feb-2022].
4. S. Paris, P. Kornprobst, J. Tumblin, and F. Durand, *A Gentle Introduction to Bilateral Filtering and its Applications*, 2007. [Online]. Available: [https://people.csail.mit.edu/sparis/bf\\_course/](https://people.csail.mit.edu/sparis/bf_course/). [Accessed: 26-Feb-2022].
5. N. Khan, S. Camalot, and S. Paris, *Lab III: Bilateral filter lab*, 2020. [Online]. Available: [https://cs.brown.edu/courses/csci1290/labs/lab\\_bilateral/index.html](https://cs.brown.edu/courses/csci1290/labs/lab_bilateral/index.html). [Accessed: 26-Feb-2022].
6. S. Paris, “Fixing the Gaussian Blur : the Bilateral Filter,” *A Gentle Introduction to Bilateral Filtering and its Applications*. [Online]. Available: [https://people.csail.mit.edu/sparis/bf\\_course/slides/03\\_definition\\_bf.pdf](https://people.csail.mit.edu/sparis/bf_course/slides/03_definition_bf.pdf). [Accessed: 26-Feb-2022].

7. S. Paris, P. Kornprobst, J. Tumblin, and F. Durand, *A Gentle Introduction to Bilateral Filtering and its Applications*, 2007. [Online]. Available: [https://people.csail.mit.edu/sparis/bf\\_course/course\\_notes.pdf](https://people.csail.mit.edu/sparis/bf_course/course_notes.pdf). [Accessed: 26-Feb-2022].
8. P. Thomas. Challenges and limitations of cloud-based ANPR solutions, 2020. [Online]. Available: <https://www.asmag.com/showpost/31736.aspx>. [Accessed: 26-Feb-2022].
9. N. Mufti & S. A. A. Shah. Automatic number plate Recognition: A detailed survey of relevant algorithms. *Sensors*, 21(9), 2021. [Online] Available : <https://www.mdpi.com/1424-8220/21/9/3028>. [Accessed: 26-Feb-2022].
10. E. Wolfart, R. Fisher, S. Perkins, and A. Walker, "Gaussian smoothing," *Spatial Filters - Gaussian Smoothing*. [Online]. Available: <https://homepages.inf.ed.ac.uk/rbf/HIPR2/gsmooth.htm>. [Accessed: 19-Mar-2022].

## Appendix

### Simple Number Plate Detection without using Deep Learning

```
from google.colab import drive
drive.mount('/content/drive')

!pip install easyocr
!pip install imutils
!pip install opencv-python-headless==4.5.2.52

import cv2
from matplotlib import pyplot as plt
import numpy as np
import imutils
import easyocr
import re

"""# 1. Read in Image, Grayscale and Blur"""

img = cv2.imread('/content/drive/MyDrive/NUS-
Y1S2/MA1508E/ANPR/images/test3.jpg')
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
plt.imshow(cv2.cvtColor(gray, cv2.COLOR_BGR2RGB))

"""# 2. Apply filter and find edges for localization"""

bfilter = cv2.bilateralFilter(gray, 11, 17, 17) # Noise reduction
```

```

edged = cv2.Canny(bfilter, 30, 200) # Edge detection
plt.imshow(cv2.cvtColor(edged, cv2.COLOR_BGR2RGB))

"""# 3. Find Contours and Apply Mask"""

keypoints = cv2.findContours(edged.copy(), cv2.RETR_TREE,
cv2.CHAIN_APPROX_SIMPLE) # first one is source image, second is contour
retrieval mode, third is contour approximation method. [SIMPLE= 4 point
detection, NONE=734 Points]
contours = imutils.grab_contours(keypoints)
contours = sorted(contours, key=cv2.contourArea, reverse=True)[:10] #
Sort from descending to ascending, get the top 10 contours
sample = keypoints[0] if len(keypoints) == 2 else keypoints[1]
keypoints[1] # These are the Hierarchies

location = None
for contour in contours:
    approx = cv2.approxPolyDP(contour, 10, True) # approximates a curve
or a polygon with another curve/polygon with less vertices so that the
distance between them is less or equal to the specified precision
    if len(approx) == 4:
        location = approx # This is the location of the number plate,
FOUND !!!
        break

"""##Contour box location"""

location # (4, 1, 2)

mask = np.zeros(gray.shape, np.uint8) # Return a new array of given
shape and type, filled with zeros. Syntax: numpy.zeros(shape,
dtype=float, order='C', *, like=None)
new_image = cv2.drawContours(mask, [location], 0,255, -1) # draw contour
lines over mask
new_image = cv2.bitwise_and(img, img, mask=mask) # The bitwise_and
operator returns an array that corresponds to the resulting image from
the merger of the given two images.

plt.imshow(cv2.cvtColor(new_image, cv2.COLOR_BGR2RGB)) #Convert an image
from one color space to another Syntax: cv2.cvtColor(src, code[, dst[,
dstCn]])

(x,y) = np.where(mask==255) # Return elements chosen from x or y
depending on condition.
x,y # shop array of detected license plate

```

```

(x1, y1) = (np.min(x), np.min(y))
(x2, y2) = (np.max(x), np.max(y))
cropped_image = gray[x1:x2+1, y1:y2+1]

plt.imshow(cv2.cvtColor(cropped_image, cv2.COLOR_BGR2RGB))

"""# 4. Use Easy OCR To Read Text"""

reader = easyocr.Reader(['en']) #set language to detect
result = reader.readtext(cropped_image) #set target for reading
result

"""# 5. Render Result"""

text = result[0][-2]
print(text)# Before: PGMN112
remove_lower = lambda text: re.sub('[a-z]', '', text)
text = remove_lower(text)

print(text)# After: PGMN112
font = cv2.FONT_HERSHEY_SIMPLEX
#display text under license plate
res = cv2.putText(img, text=text, org=(approx[0][0][0],
approx[1][0][1]+60), fontFace=font, fontScale=1, color=(0,255,0),
thickness=2, lineType=cv2.LINE_AA)
res = cv2.rectangle(img, tuple(approx[0][0]), tuple(approx[2][0]),
(0,255,0),3)
plt.imshow(cv2.cvtColor(res, cv2.COLOR_BGR2RGB))

```

#### Grayscale conversion

```

import numpy as np
import cv2
import matplotlib.pyplot as plt

image_path = 'red_rose.jpg'
image1 = cv2.imread(image_path, cv2.IMREAD_COLOR)
image2 = cv2.cvtColor(image1, cv2.COLOR_BGR2GRAY) # RGB to Gray
print(image2)
image2 = cv2.resize(image2, (100,100))
image1.shape

```

```

# Step 1
for row in range(400,600):
    for column in range(400,600):
        image1[row][column] = image1[row][column][0] * 0.114 +
image1[row][column][1] * 0.587 + image1[row][column][2] * 0.299

plt.imshow(image1)
plt.title('img_read_as_color')
plt.show()

```

### Bilateral Filtering

```

import numpy as np
import cv2
import matplotlib.pyplot as plt
from functools import lru_cache

# Gaussian convolution applied at each pixel in an image
# Parameters:
#   im: Image to filter, provided as a numpy array
#   sigma: The variance of the Gaussian (e.g., 1)

@lru_cache(maxsize=None)
def gaussian(path, sigma):
    im = cv2.imread(path, cv2.IMREAD_COLOR)
    im = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY) # COLOR_BGR2GRAY
    height, width = im.shape
    img_filtered = np.zeros([height, width, 3])

    # Define filter size.
    # A Gaussian has infinite support, but most of it's mass lies within
    three standard deviations of the mean.
    # The standard deviation is the square of the variance, sigma.
    n = np.int(np.sqrt(sigma) * 3)

    # Iterate over pixel locations p
    for p_y in range(height):

```

```

for p_x in range(width):
    gp = 0
    w = 0

    # Iterate over kernel locations to define pixel q
    for i in range(-n, n):
        for j in range(-n, n):
            # Make sure no index goes out of bounds of the image
            q_y = np.max([0, np.min([height - 1, p_y + i])])
            q_x = np.max([0, np.min([width - 1, p_x + j])])

            constant = 1 / (2 * np.pi * sigma ** 2)
            # print("px = {0}, py = {1}, qx = {2}, qy =
{3}".format(p_x, p_y, q_x, q_y))

            # Compute Gaussian filter weight at this filter
pixel
            g = constant * np.exp(-((q_x - p_x)**2 + (q_y -
p_y)** 2) / (2 * sigma**2))
            # print("g = {}".format(g))

            gr_x = abs(im[p_y, p_x] - im[q_y, q_x])
            # print("gr_x = {}".format(gr_x))

            gr = constant * np.exp(-(gr_x ** 2) / (2 *
sigma**2))
            # print("gr = {}".format(gr))

            # Accumulate filtered output
            gp += g * im[p_y, p_x] * gr

            # Accumulate filter weight for later normalization,
to maintain image brightness
            w += g
            img_filtered[p_y, p_x, :] = gp / (w +
np.finfo(np.float32).eps) # eps = epsilon
        return img_filtered

img_filtered = gaussian(path="img\\rubiks_cube.png", sigma=1)
plt.imshow(img_filtered)

```

Link to Simulation : <https://colab.research.google.com/drive/1W>

Commented [1]: Will update before submission.