1. Objectives

This activity aims to introduce students to object detection and recongition through the use of histogram of gradients (HoG), image pyramids and sliding windows in OpenCV.

2. Intended Learning Outcomes (ILOs)

After this activity, the students should be able to:

- Demonstrate object detection and recognition using histogram of gradients, image pyramids and sliding windows in OpenCV.
- Evaluate the performance of the object detection and recognition techniques in OpenCV.

3. Procedures and Outputs

When it comes to recognizing and detecting objects, there are a number of techniques used in computer vision, which we'll be examining:

- Histogram of Oriented Gradients
- Image pyramids
- Sliding windows

Unlike feature detection algorithms, these are not mutually exclusive techniques, rather, they are complimentary. You can perform a Histogram of Oriented Gradients (HOG) while applying the sliding windows technique. So, let's take a look at HOG first and understand what it is.

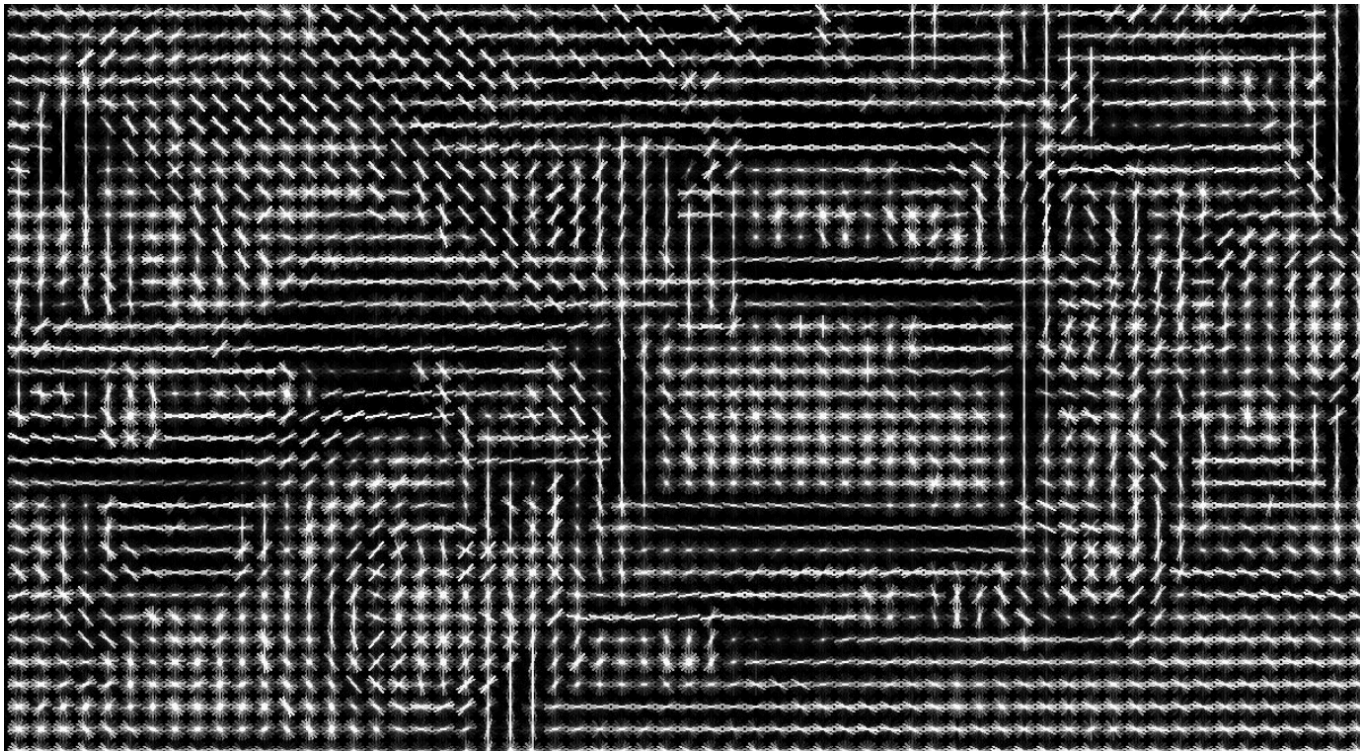
History of Gradients (HoG) Descriptors

HOG is a feature descriptor, so it belongs to the same family of algorithms, such as SIFT, SURF, and ORB.

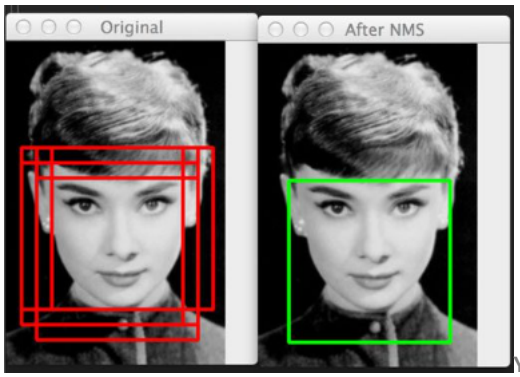
It is used in image and video processing to detect objects. Its internal mechanism is really clever; an image is divided into portions and a gradient for each portion is calculated. We've observed a similar approach when we talked about face recognition through LBPH.

HOG, however, calculates histograms that are not based on color values, rather, they are based on gradients. As HOG is a feature descriptor, it is capable of delivering the type of information that is vital for feature matching and object detection/recognition.

Before diving into the technical details of how HOG works, let's first take a look at how HOG sees the world; here is an image of a truck:



Question: Here you are shown the original image and how HOG sees the image. Analyze the output and provide a description. Are you able to see that it is the same image?

- 

Yes, the original image and HOG representation are similar. The HOG image shows similar patterns and texture from the original image.

The extrapolation of histograms into descriptors is quite a complex process.

- First, local histograms for each cell are calculated. The cells are grouped into larger regions called blocks. These blocks can be made of any number of cells, but Dalal and Triggs found that 2x2 cell blocks yielded the best results when performing people detection.
- A block-wide vector is created so that it can be normalized, accounting for variations in illumination and shadowing (a single cell is too small a region to detect such variations). This improves the accuracy of detection as it reduces the illumination and shadowing difference between the sample and the block being examined.
- Simply comparing cells in two images would not work unless the images are identical (both in terms of size and data).

There are two main problems to resolve:

1. Location
2. Scale

The Scale Issue

Imagine, for example, if your sample was a detail (say, a bike) extrapolated from a larger image, and you're trying to compare the two pictures. You would not obtain the same gradient signatures and the detection would fail (even though the bike is in both pictures).

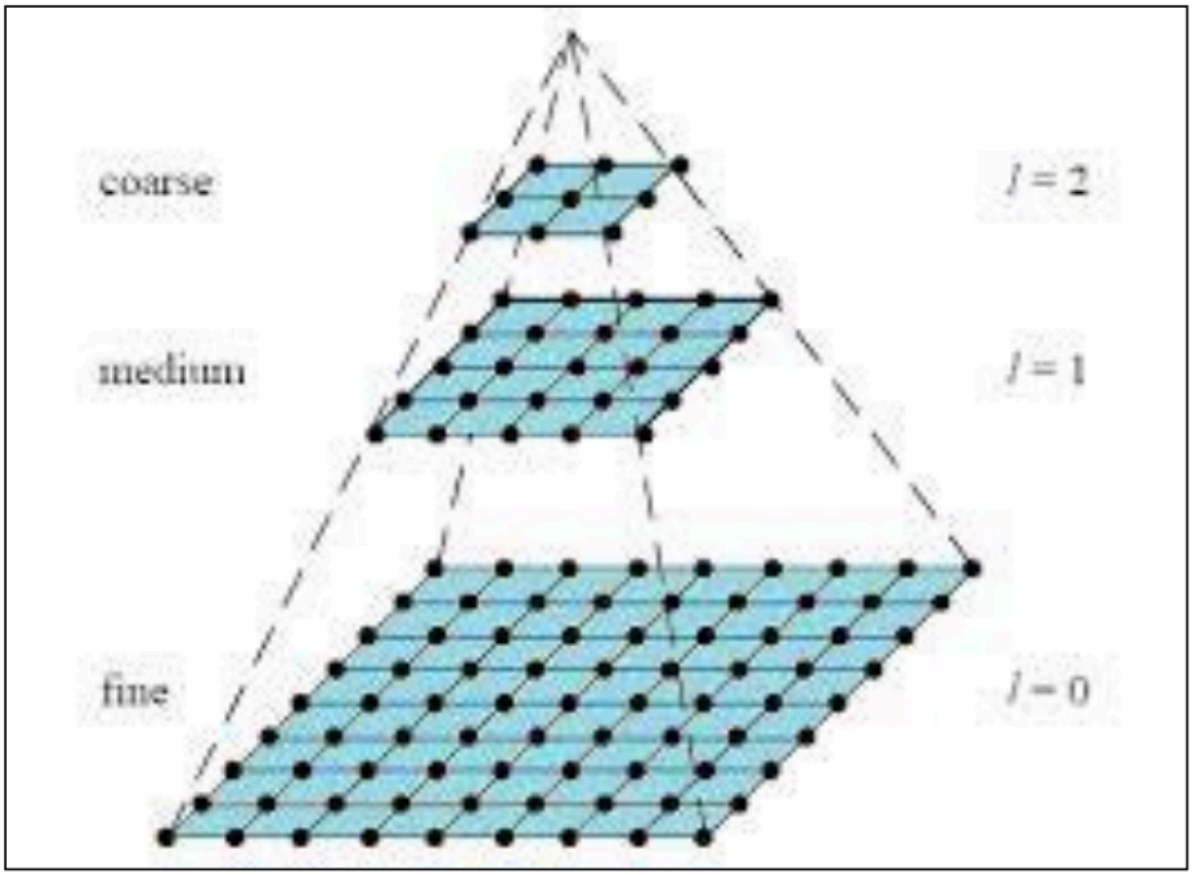
The Location Issue

Once we've resolved the scale problem, we have another obstacle in our path: a potentially detectable object can be anywhere in the image, so we need to scan the entire image in portions to make sure we can identify areas of interest, and within these areas, try to detect objects. Even if a sample image and object in the image are of identical size, there needs to be a way to instruct OpenCV to locate this object. So, the rest of the image is discarded and a comparison is made on potentially matching regions.

To obviate these problems, we need to familiarize ourselves with the concepts of image pyramid and sliding windows.

Image Pyramids

Many of the algorithms used in computer vision utilize a concept called pyramid. An image pyramid is a multiscale representation of an image. This diagram should help you understand this concept:



A multiscale representation of an image, or an image pyramid, helps you resolve the problem of detecting objects at different scales. The importance of this concept is easily explained through real-life hard facts, such as it is extremely unlikely that an object will appear in an image at the exact scale it appeared in our sample image. Moreover, you will learn that object classifiers (utilities that allow you to detect objects in OpenCV) need training, and this training is provided through image databases made up of positive matches and negative matches. Among the positives, it is again unlikely that the object we want to identify will appear in the same scale throughout the training dataset.

Follow the steps and create a function that can make an image Pyramid:

1. Take an image.
2. Resize (smaller) the image using an arbitrary scale parameter.
3. Smoothen the image (using Gaussian blurring).
4. If the image is larger than an arbitrary minimum size, repeat the process from step 1.

Demonstrate by creating a function and showing the different scales of the image.

```
from google.colab.patches import cv2_imshow
import cv2
import numpy as np
import matplotlib.pyplot as plt
```

```
# Load the image
image = cv2.imread("/content/minnie.jpg")

# Resize the image using an arbitrary scale parameter
scale = 0.5
resized_image = cv2.resize(image, None, fx=scale, fy=scale)

# Smoothen the image using Gaussian blurring
blurred_image = cv2.GaussianBlur(resized_image, (5, 5), 0)

# Arbitrary minimum size
min_size = (30, 30)

while resized_image.shape[0] > min_size[0] and resized_image.shape[1] > min_size[1]:
    cv2_imshow(blurred_image)
    cv2.waitKey(0)

    # Resize the image again
    resized_image = cv2.resize(resized_image, None, fx=scale, fy=scale)

    # Apply Gaussian blur again
    blurred_image = cv2.GaussianBlur(resized_image, (5, 5), 0)

cv2.destroyAllWindows()
```

Question: How does the performance of your functions in obtaining the image pyramid perform in iterations of the image that decrease until the scale factor is achieved?

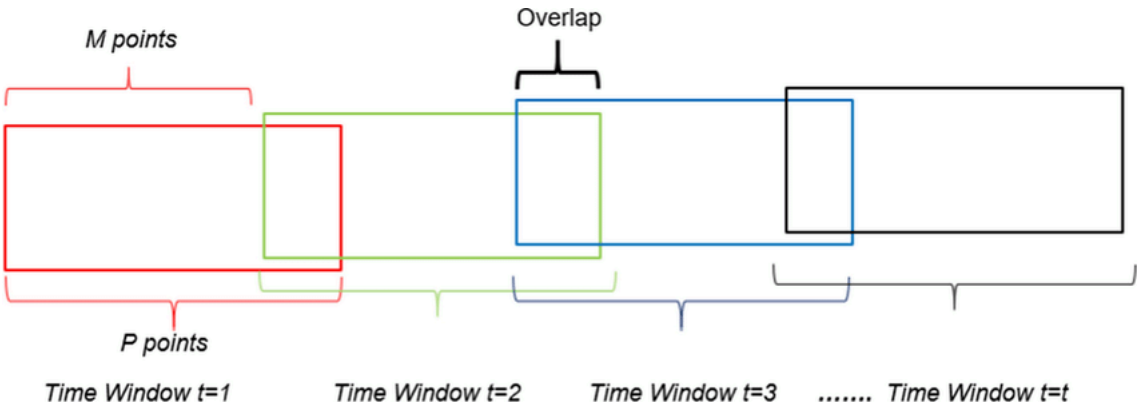
Sliding Windows

Sliding windows is a technique used in computer vision that consists of examining the shifting portions of an image (sliding windows) and operating detection on those using image pyramids. This is done so that an object can be detected at a multiscale level.

Sliding windows resolves location issues by scanning smaller regions of a larger image, and then repeating the scanning on different scales of the same image.

With this technique, each image is decomposed into portions, which allows discarding portions that are unlikely to contain objects, while the remaining portions are classified.

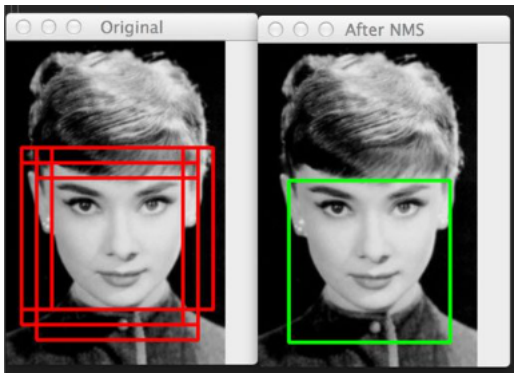
There is one problem that emerges with this approach, though: **overlapping regions**.



Here's where non-maximum suppression comes into play: given a set of overlapping regions, we can suppress all the regions that are not classified with the maximum score.

Question: Research on the topic and provide ample discussion, what is non-maximum (or non-maxima) suppression? Provide examples and code.

- Non-maximum suppression is a teacnique commonly used in computer vision and object detection task. It is a class algorithm that eliminates overlapping bounding boxes or region of interest. This method use a moving window over through the feature map to assign background scored basef on th efeatures calculated in the window. It works by iterating over all the detected bounding boxes with a highest confident score



The general approach behind non-maximum suppression is as follows:

1. Once an image pyramid has been constructed, scan the image with the sliding window approach for object detection.
2. Collect all the current windows that have returned a positive result (beyond a certain arbitrary threshold), and take a window, W, with the highest response.

3. Eliminate all windows that overlap W significantly.
4. Move on to the next window with the highest response and repeat the process for the current scale.

Question: Demonstrate the above given steps using a custom image and creating code in Python using OpenCV

```
from google.colab.patches import cv2_imshow
import cv2
import numpy as np
import matplotlib.pyplot as plt

image = cv2.imread("/content/minnie.jpg")

rectangle = image
stepSize = 40
(w_width, w_height) = (30, 30)
for x in range(0, image.shape[1] - w_width , stepSize):
   for y in range(0, image.shape[0] - w_height, stepSize):
      window = image[x:x + w_width, y:y + w_height, :]

      cv2.rectangle(rectangle, (x, y), (x + w_width, y + w_height), (0, 255, 0), 1)
      plt.imshow(np.array(rectangle).astype('uint8'))

plt.show()



# Load the custom image
image = cv2.imread("/content/minnie.jpg")


detected_windows = [((100, 100), (200, 200), 0.9),
                    ((150, 150), (250, 250), 0.8),
                    ((200, 200), (300, 300), 0.7)]


highest_confidence_window = detected_windows[0]


threshold = 0.5  # Arbitrary threshold for significant overlap
selected_windows = [highest_confidence_window]

for window in detected_windows[1:]:

    x_tl, y_tl = window[0]
    x_br, y_br = window[1]
    x_tl_h, y_tl_h = highest_confidence_window[0]
    x_br_h, y_br_h = highest_confidence_window[1]

    intersection_area = max(0, min(x_br, x_br_h) - max(x_tl, x_tl_h)) * max(0, min(y_br, y_br_h) - max(y_tl, y_tl_h))
    area_current_window = (x_br - x_tl) * (y_br - y_tl)
    area_highest_confidence_window = (x_br_h - x_tl_h) * (y_br_h - y_tl_h)
    union_area = area_current_window + area_highest_confidence_window - intersection_area
    iou = intersection_area / union_area


    if iou < threshold:
        selected_windows.append(window)


result_image = image.copy()
for window in selected_windows:
    cv2.rectangle(result_image, window[0], window[1], (0, 255, 0), 2)

cv2_imshow(result_image)
cv2.waitKey(0)
cv2.destroyAllWindows()
```