ARTIFICIAL INTELLIGENCE SEMESTER PROJECT

SUBMITTED BY:-

YASIR ALI LASHARI

ABDUL GHANI LARIK

INTRODUCTION OF YOLO V4

YOLO is an abbreviation for the term ‘You Only Look Once’. This is an algorithm that detects and recognizes various objects in a picture (in real-time). Object detection in YOLO is done as a regression problem and provides the class probabilities of the detected images. YOLO algorithm employs convolutional neural networks (CNN) to detect objects in real-time. As the name suggests, the algorithm requires only a single forward propagation through a neural network to detect objects. This means that prediction in the entire image is done in a single algorithm run. The CNN is used to predict various class probabilities and bounding boxes simultaneously.

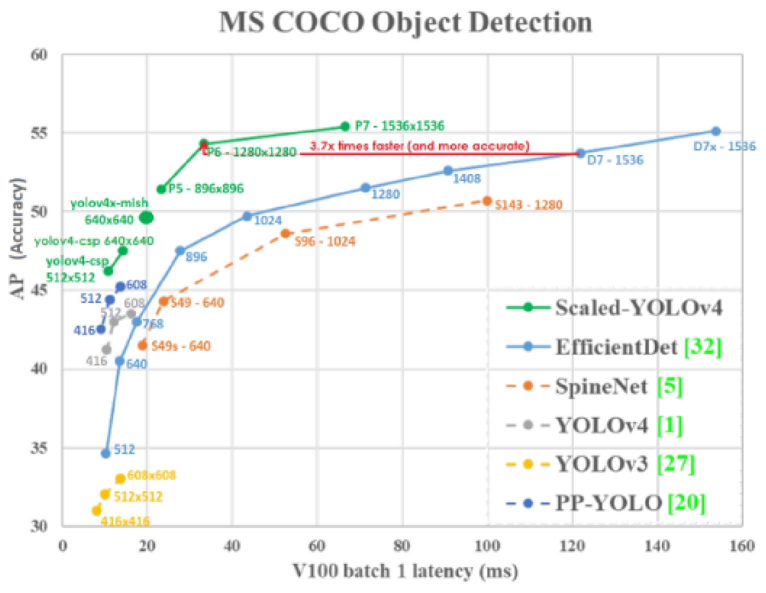
Here we used Yolo version 4 Actually YOLOv4 is known for its up-gradation in terms of AP and FPS. YOLOv4 prioritizes real-time object detection and training takes place on a single CPU. YOLOv4 has obtained state-of-art results on the COCO dataset with 43.5% speed (AP) at 65 Performance (FPS) on Tesla V100. This achievement is the result of a combination of the features like Drop Block Regularization, Data Augmentation, Mish-Activation, Cross Stage-Partial-connections (CSP), Self-adversarial-training (SAT), Weighted-Residual-Connections (WRC) and many more.

WORKING

We are going to perform real-time Yolo version 4 object detections on my webcam within google collab we'll be doing real-time object detections on both images and video within google collab.

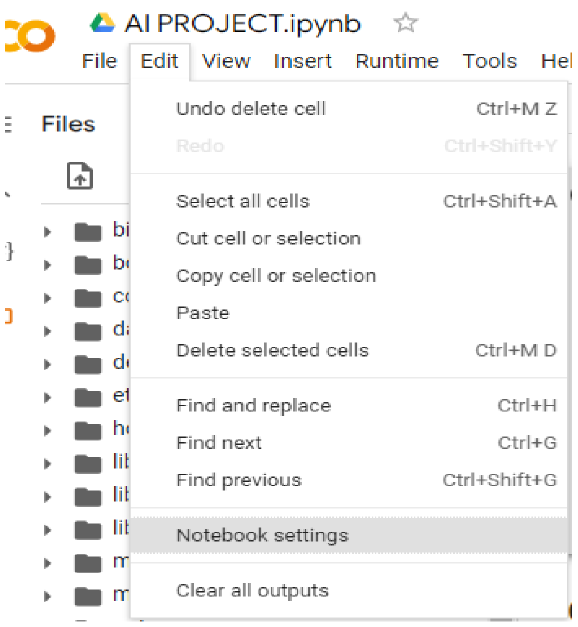
YOLOv4 Object Detection on Webcam In Google Colab

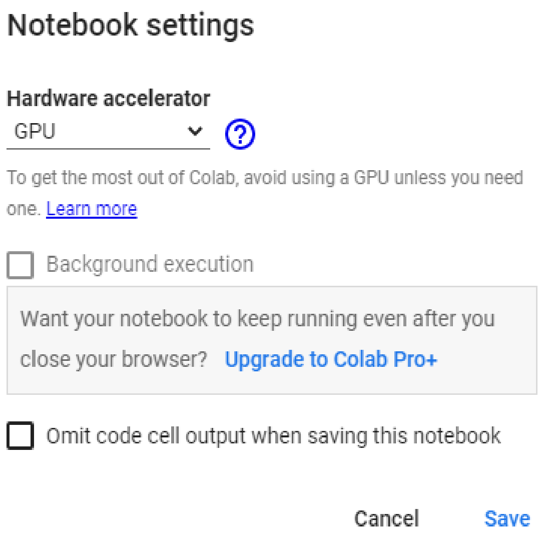
This notebook will walkthrough all the steps for performing YOLOv4 object detections on your webcam while in Google Colab. We will be using scaled-YOLOv4 (yolov4-csp) for this tutorial, the fastest and most accurate object detector there currently is.



we will be using scaled Yolo version 4 which is the latest version of Yolo v4 and is the most fast and accurate object detector there is right now out there so more specifically we're going to be using Yolo v4 csp 640x640 it'll allow us to run our real-time Yolo version 4 on live streaming video from our webcam.

firstly we're going to want to go to edit right here in the top left and go to notebook settings and make sure that our hardware accelerator is set to gpu and hit save this is going to allow us to actually do the Yolo version 4 detections in real time otherwise if we don't enable gpu we're going to be running on cpu and have much slower detections.




Notebook settings

Hardware accelerator
GPU

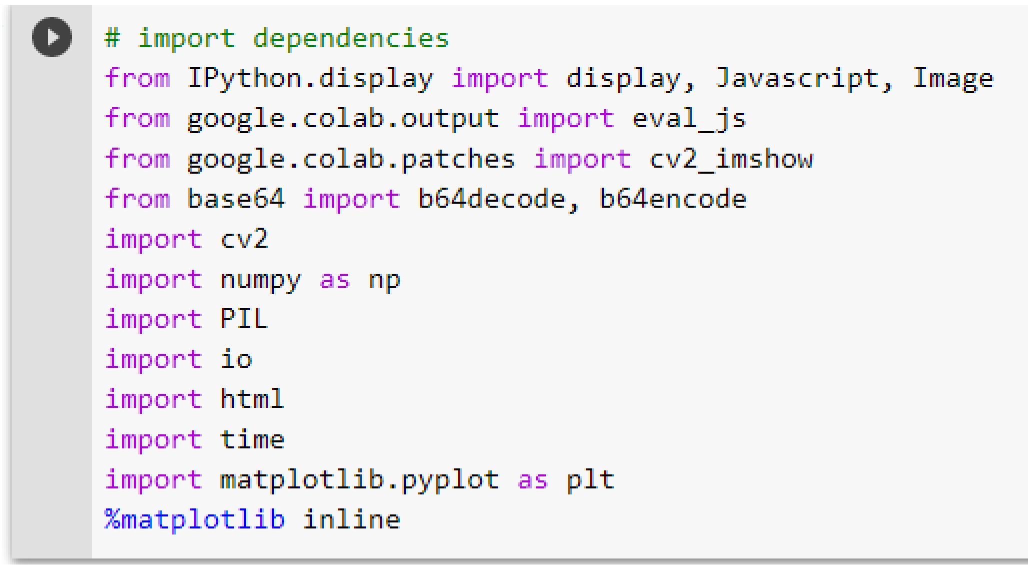
To get the most out of Colab, avoid using a GPU unless you need one. Learn more

☐ Background execution

Want your notebook to keep running even after you close your browser? Upgrade to Colab Pro+

☐ Omit code cell output when saving this notebook

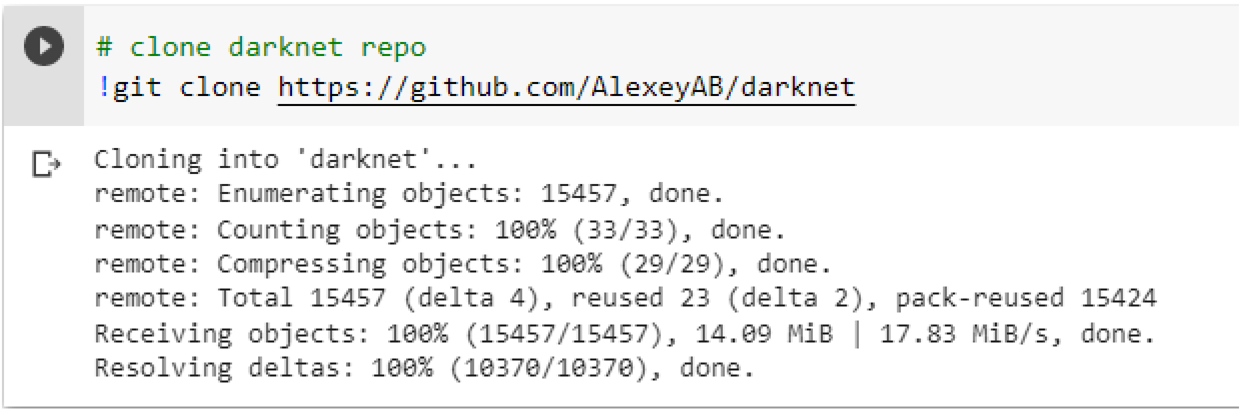
After that we're just going to go ahead and import our dependencies that's going to download all of the required files and packages.



```python
# import dependencies
from IPython.display import display, Javascript, Image
from google.colab.output import eval_js
from google.colab.patches import cv2_imshow
from base64 import b64decode, b64encode
import cv2
import numpy as np
import PIL
import io
import html
import time
import matplotlib.pyplot as plt
%matplotlib inline
```

After that

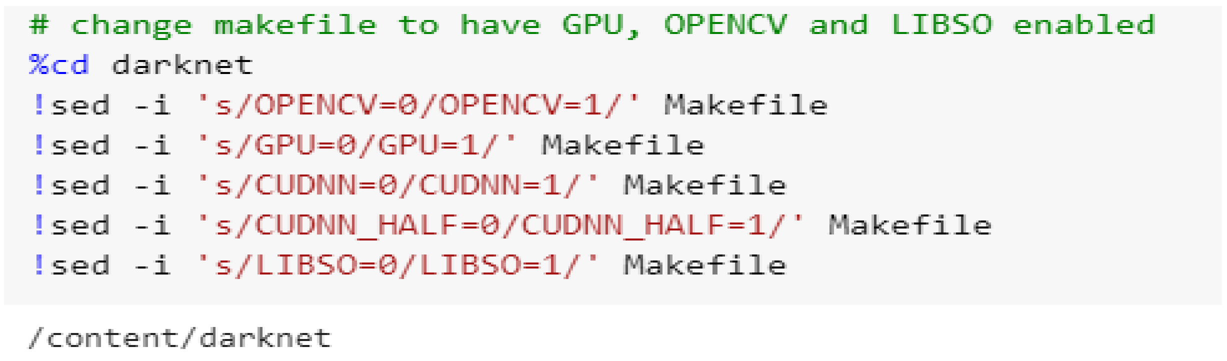
We will be using the famous AlexeyAB's darknet repository to perform YOLOv4 detections.



```
# clone darknet repo
!git clone https://github.com/AlexeyAB/darknet
```

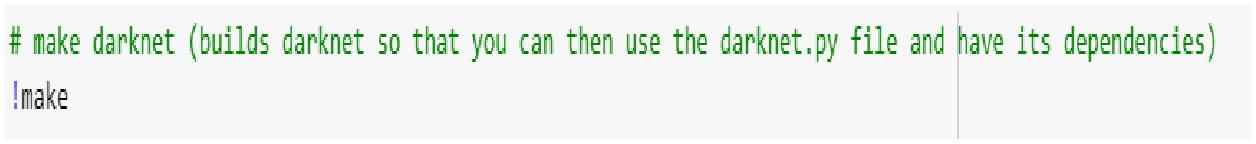
```
Cloning into 'darknet'...
remote: Enumerating objects: 15457, done.
remote: Counting objects: 100% (33/33), done.
remote: Compressing objects: 100% (29/29), done.
remote: Total 15457 (delta 4), reused 23 (delta 2), pack-reused 15424
Receiving objects: 100% (15457/15457), 14.09 MiB | 17.83 MiB/s, done.
Resolving deltas: 100% (10370/10370), done.
```

then it's just going to step into the repository and edit the make file to enable gpu and opencv as well as this libso is going to allow us to actually get access to darknet using python. Additionally we're going to go ahead and run the make command this is going to build all the binaries files and get darknet properly running so that we can access the darknet.pi file and run python functions in order to have the Yolo v4 detections.



```
# change makefile to have GPU, OPENCV and LIBSO enabled
%cd darknet
!sed -i 's/OPENCV=0/OPENCV=1/' Makefile
!sed -i 's/GPU=0/GPU=1/' Makefile
!sed -i 's/CUDNN=0/CUDNN=1/' Makefile
!sed -i 's/CUDNN_HALF=0/CUDNN_HALF=1/' Makefile
!sed -i 's/LIBSO=0/LIBSO=1/' Makefile
```

```
/content/darknet
```

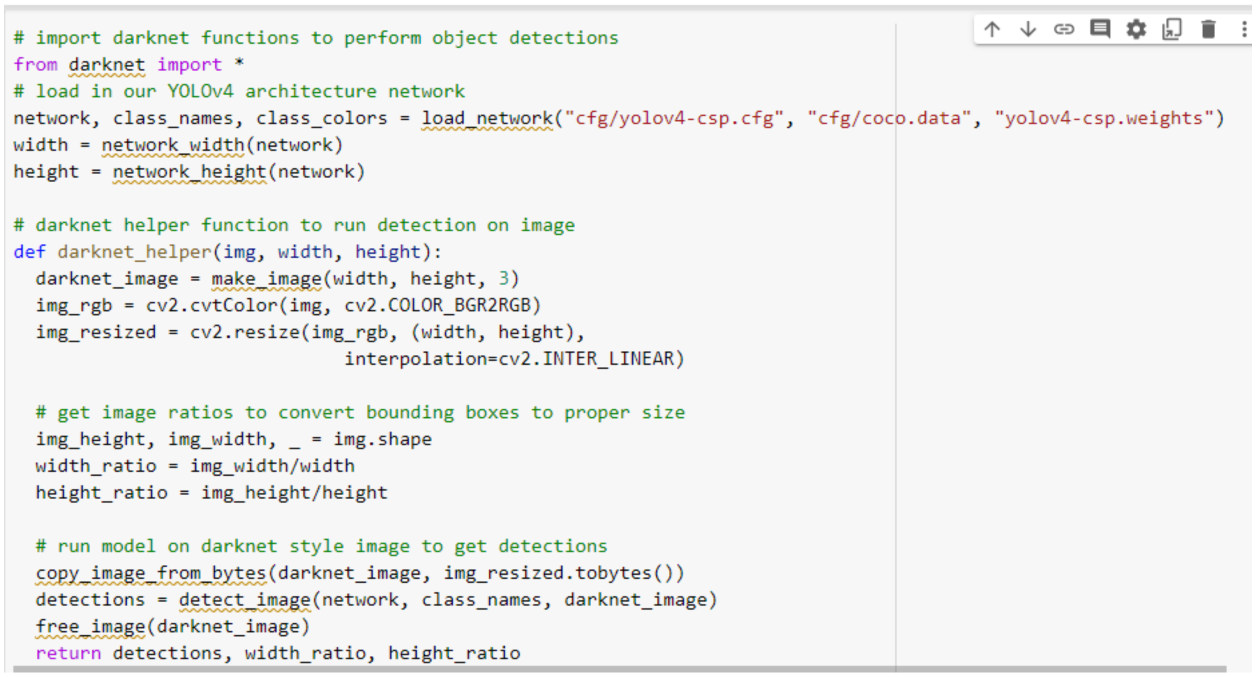


```
# make darknet (builds darknet so that you can then use the darknet.py file and have its dependencies)
!make
```

Then we're just going to go to the next cell which is this w get what this is doing is it's grabbing the scaled Yolo v4 weights file the model file that's pre-trained on over   classes or different objects that it can detect which we will utilize but if we wanted to use one custom trained in this spot we can do that just upload our custom trained model file.

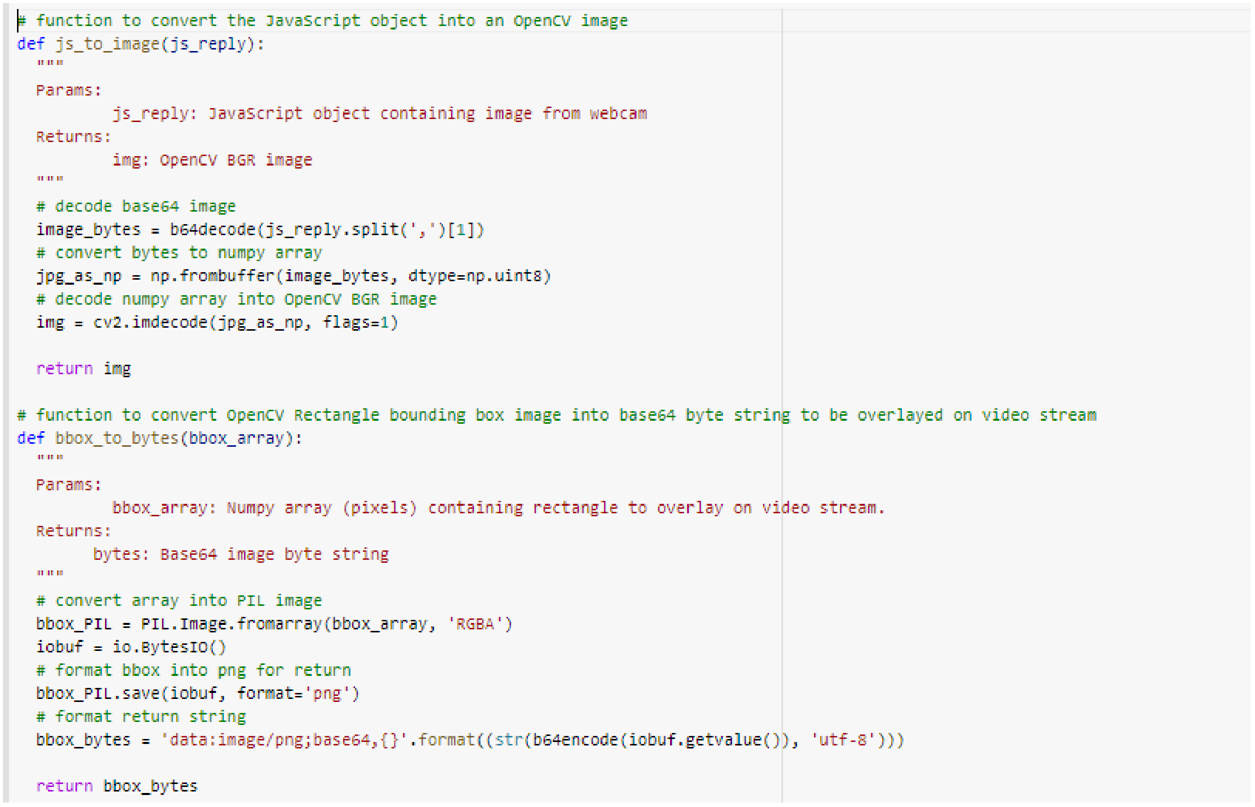
```
!wget --load-
cookies /tmp/cookies.txt "https://docs.google.com/uc?export=downloa
d&confirm=$(wget --quiet --save-cookies /tmp/cookies.txt --keep-
session-cookies --no-check-
certificate 'https://docs.google.com/uc?export=download&id=1V3vsIax
AlGWvK4Aar9bAiK5U0QFttKwq' -O- | sed -rn 's/.*confirm=([0-9A-Za-
z_]+).*/\1\n/p')&id=1V3vsIaxAlGWvK4Aar9bAiK5U0QFttKwq" -O yolov4-
csp.weights && rm -rf /tmp/cookies.txt
```

Here we're just going to load in our model right here using the load network function from the darknet.pi so then there's this darknet helper function that just passes in the image the width and the height of the image and then it's going to go ahead and run the detect image command from the darknet.pi file on the image and get us those detections so it does all of the preprocessing of the image rescaling it and then gets a ratio for the bounding boxes and actually does the detections and then returns them.



```
# import darknet functions to perform object detections
from darknet import *
# load in our YOLOv4 architecture network
network, class_names, class_colors = load_network("cfg/yolov4-csp.cfg", "cfg/coco.data", "yolov4-csp.weights")
width = network_width(network)
height = network_height(network)

# darknet helper function to run detection on image
def darknet_helper(img, width, height):
  darknet_image = make_image(width, height, 3)
  img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
  img_resized = cv2.resize(img_rgb, (width, height),
                              interpolation=cv2.INTER_LINEAR)

  # get image ratios to convert bounding boxes to proper size
  img_height, img_width, _ = img.shape
  width_ratio = img_width/width
  height_ratio = img_height/height

  # run model on darknet style image to get detections
  copy_image_from_bytes(darknet_image, img_resized.tobytes())
  detections = detect_image(network, class_names, darknet_image)
  free_image(darknet_image)
  return detections, width_ratio, height_ratio
```

Here are a few helper functions defined that will be used to easily convert between different image types within our later steps.

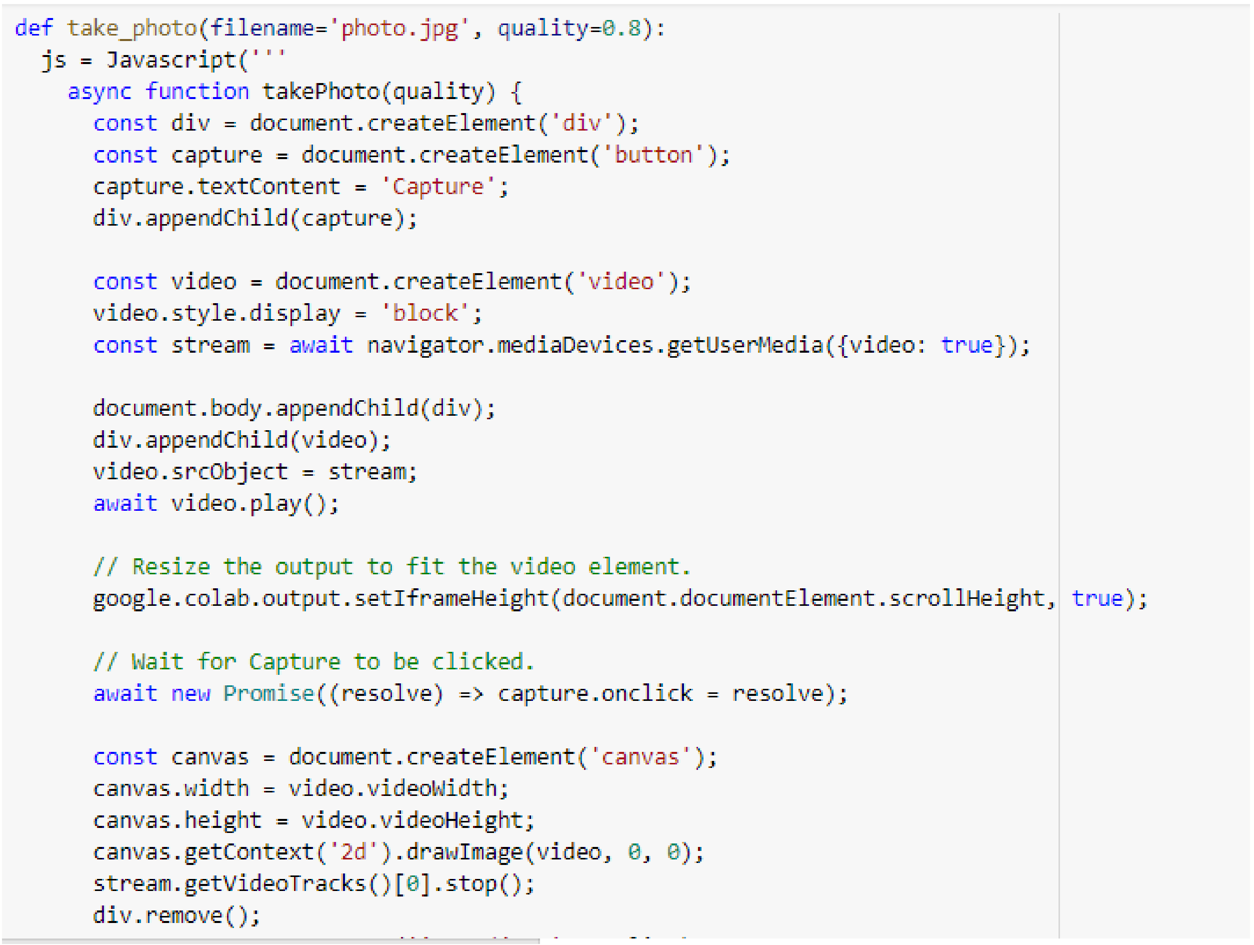


```
# function to convert the JavaScript object into an OpenCV image
def js_to_image(js_reply):
  """
  Params:
          js_reply: JavaScript object containing image from webcam
  Returns:
          img: OpenCV BGR image
  """
  # decode base64 image
  image_bytes = b64decode(js_reply.split(',')[1])
  # convert bytes to numpy array
  jpg_as_np = np.frombuffer(image_bytes, dtype=np.uint8)
  # decode numpy array into OpenCV BGR image
  img = cv2.imdecode(jpg_as_np, flags=1)

  return img

# function to convert OpenCV Rectangle bounding box image into base64 byte string to be overlayed on video stream
def bbox_to_bytes(bbox_array):
  """
  Params:
          bbox_array: Numpy array (pixels) containing rectangle to overlay on video stream.
  Returns:
        bytes: Base64 image byte string
  """
  # convert array into PIL image
  bbox_PIL = PIL.Image.fromarray(bbox_array, 'RGBA')
  iobuf = io.BytesIO()
  # format bbox into png for return
  bbox_PIL.save(iobuf, format='png')
  # format return string
  bbox_bytes = 'data:image/png;base64,{}'.format((str(b64encode(iobuf.getvalue()), 'utf-8')))

  return bbox_bytes
```

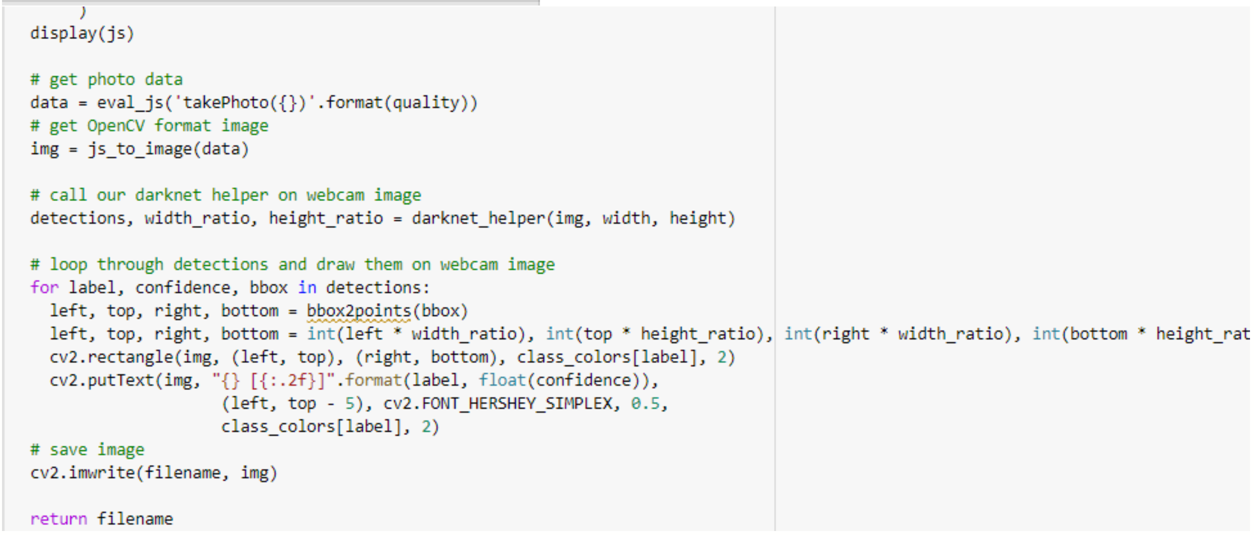
now we can go ahead to the webcam part Running YOLOv4 on images taken from webcam is fairly straight-forward. We will utilize code within Google Colab's **Code Snippets** that has a variety of useful code functions to perform various tasks.

We will be using the code snippet for **Camera Capture** which runs JavaScript code to utilize your computer's webcam. The code snippet will take a webcam photo, which we will then pass into our YOLOv4 model for object detection.

Below is a function to take the webcam picture using JavaScript and then run YOLOv4 on it.

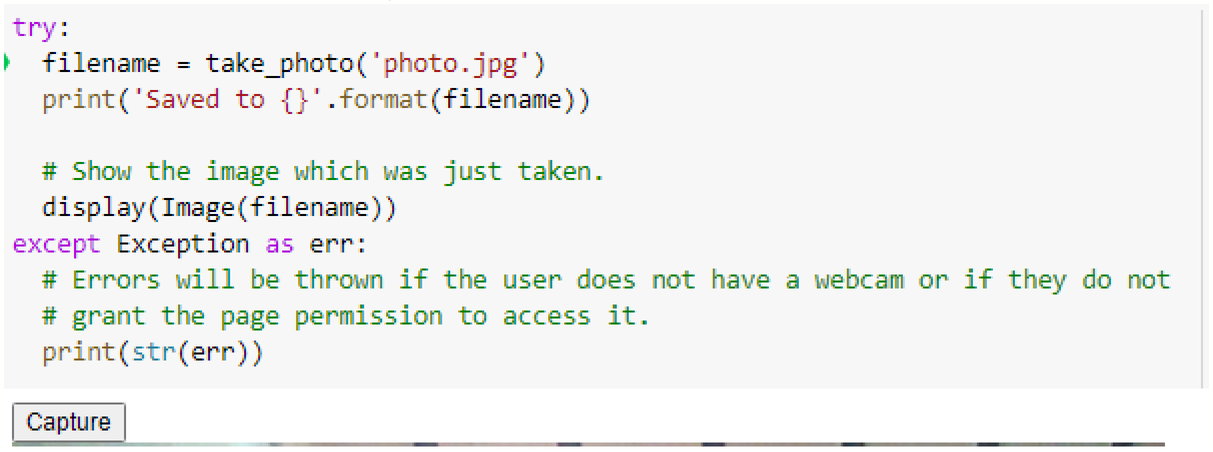


```
def take_photo(filename='photo.jpg', quality=0.8):
  js = Javascript('''
    async function takePhoto(quality) {
      const div = document.createElement('div');
      const capture = document.createElement('button');
      capture.textContent = 'Capture';
      div.appendChild(capture);

      const video = document.createElement('video');
      video.style.display = 'block';
      const stream = await navigator.mediaDevices.getUserMedia({video: true});

      document.body.appendChild(div);
      div.appendChild(video);
      video.srcObject = stream;
      await video.play();

      // Resize the output to fit the video element.
      google.colab.output.setIframeHeight(document.documentElement.scrollHeight, true);

      // Wait for Capture to be clicked.
      await new Promise((resolve) => capture.onclick = resolve);

      const canvas = document.createElement('canvas');
      canvas.width = video.videoWidth;
      canvas.height = video.videoHeight;
      canvas.getContext('2d').drawImage(video, 0, 0);
      stream.getVideoTracks()[0].stop();
      div.remove();
```



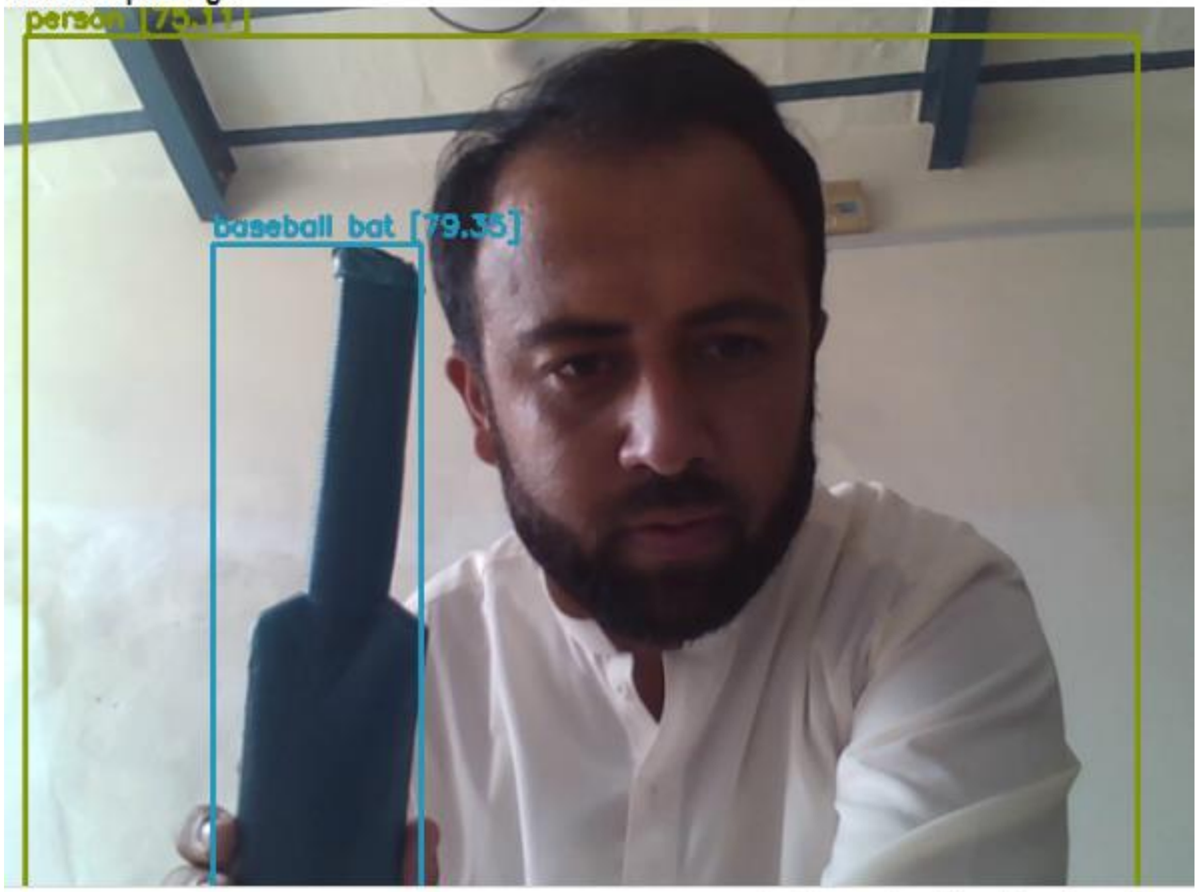
```
    )
  display(js)

  # get photo data
  data = eval_js('takePhoto({})'.format(quality))
  # get OpenCV format image
  img = js_to_image(data)

  # call our darknet helper on webcam image
  detections, width_ratio, height_ratio = darknet_helper(img, width, height)

  # loop through detections and draw them on webcam image
  for label, confidence, bbox in detections:
    left, top, right, bottom = bbox2points(bbox)
    left, top, right, bottom = int(left * width_ratio), int(top * height_ratio), int(right * width_ratio), int(bottom * height_rat
    cv2.rectangle(img, (left, top), (right, bottom), class_colors[label], 2)
    cv2.putText(img, "{} [{:.2f}]".format(label, float(confidence)),
                      (left, top - 5), cv2.FONT_HERSHEY_SIMPLEX, 0.5,
                      class_colors[label], 2)
  # save image
  cv2.imwrite(filename, img)

  return filename
```

if we're ready to run it ourwebcam on images all we do is run this cell right here so go ahead and run it should pop open ourwebcam image along with the capture button and all we're going to want to do is when we're ready to run it we just hold up whatever objects we want to detect and then hit capture.

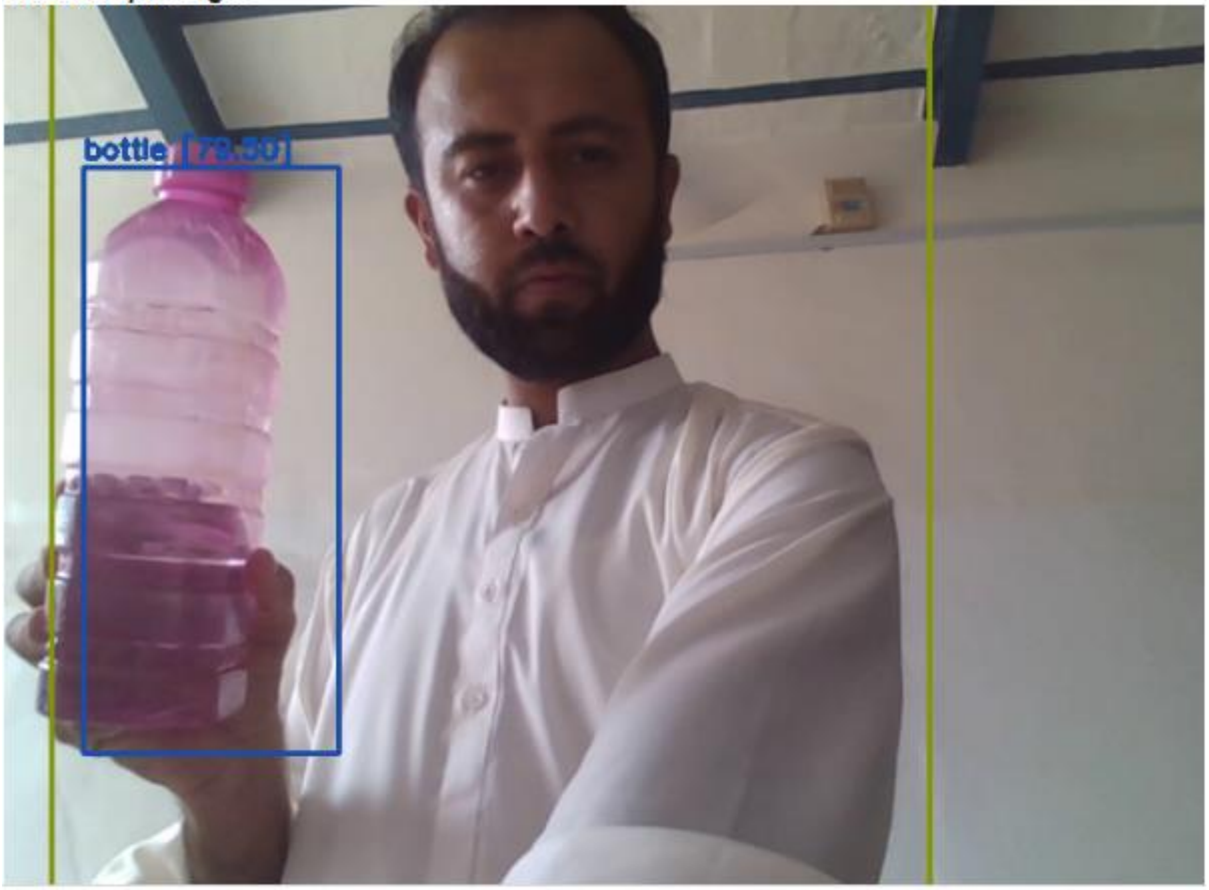


```
try:
  filename = take_photo('photo.jpg')
  print('Saved to {}'.format(filename))

  # Show the image which was just taken.
  display(Image(filename))
except Exception as err:
  # Errors will be thrown if the user does not have a webcam or if they do not
  # grant the page permission to access it.
  print(str(err))
```

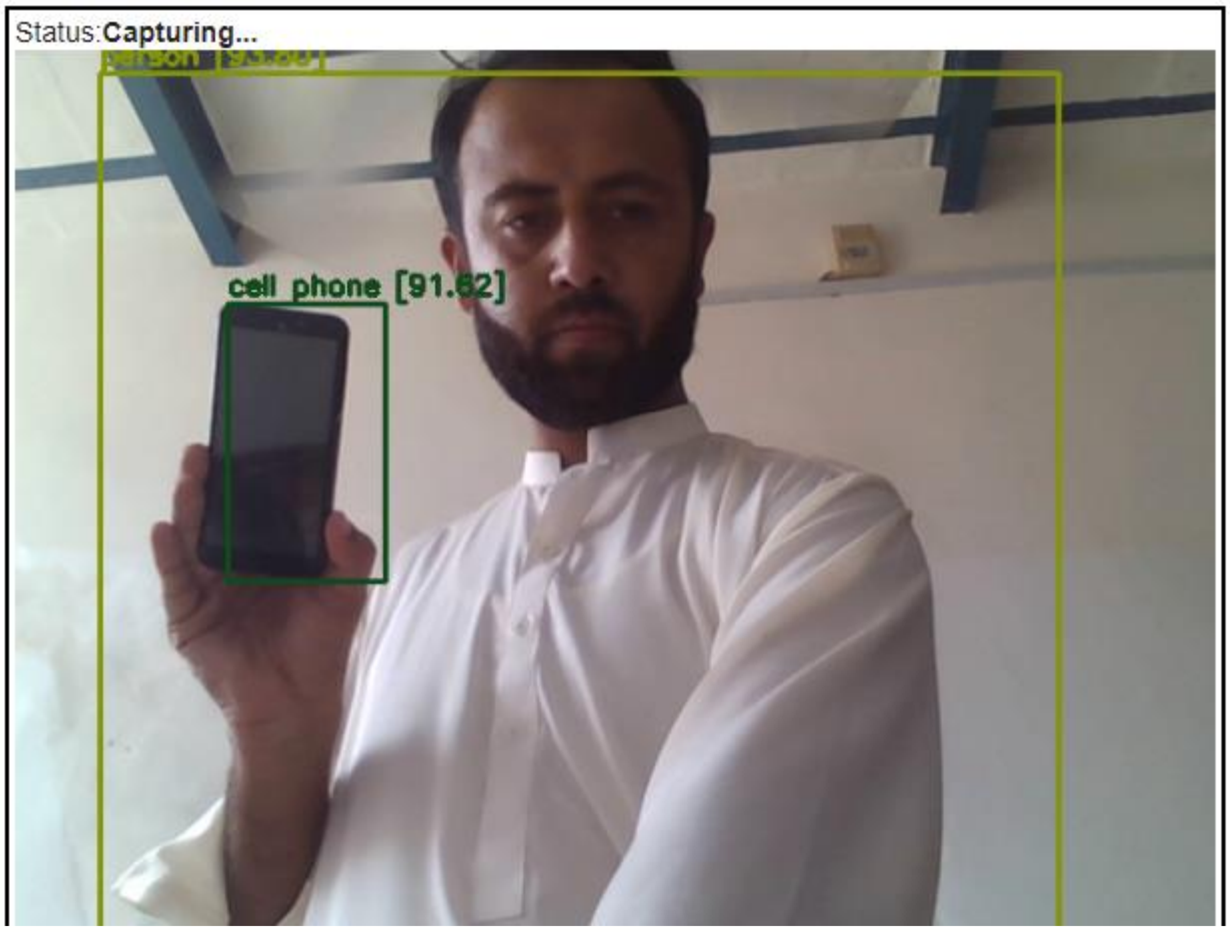
Status:**Capturing...**



Status:**Capturing...**



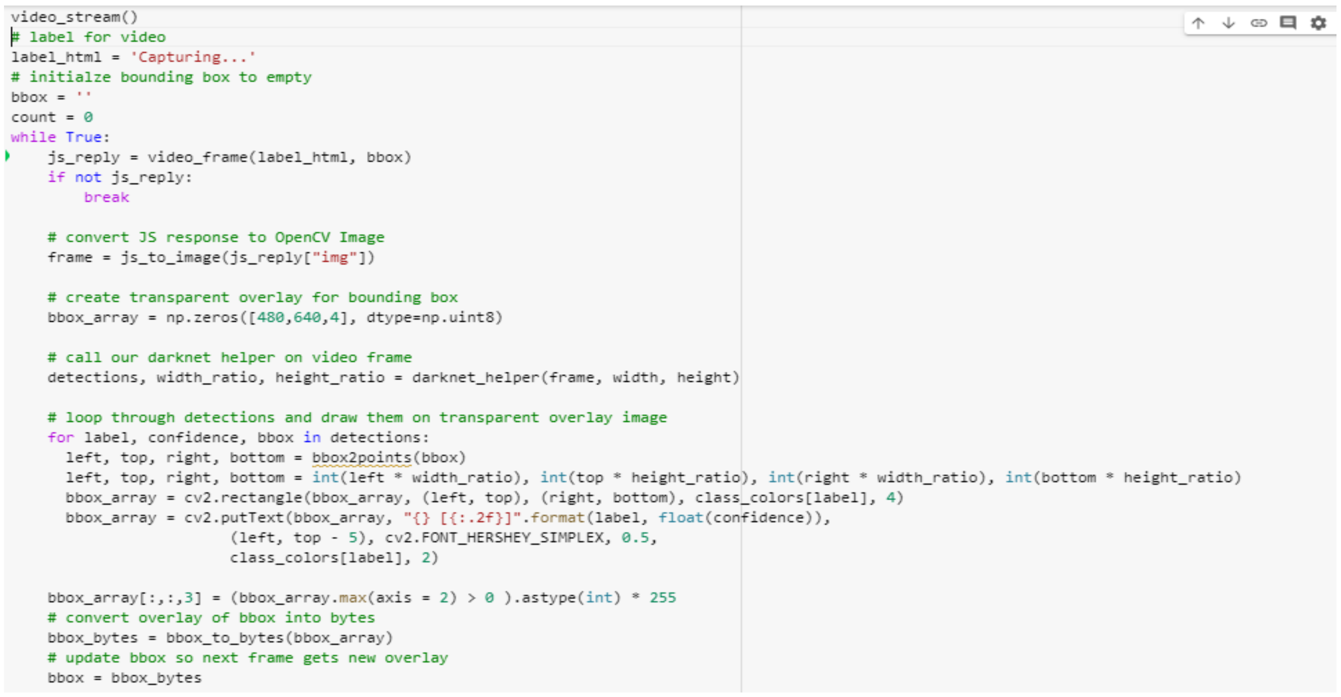
[illegible]



so now let's get on to video Running YOLOv4 on webcam video is a little more complex than images. We need to start a video stream using our webcam as input. Then we run each frame through our YOLOv4 model and create an overlay image that contains bounding box of detection(s). We then overlay the bounding box image back onto the next frame of our video stream.

YOLOv4 is so fast that it can run the detections in real-time!

Below is a function to start up the video stream using similar JavaScript as was used for images. The video stream frames are fed as input to YOLOv4.



```
video_stream()
# label for video
label_html = 'Capturing...'
# initialze bounding box to empty
bbox = ''
count = 0
while True:
    js_reply = video_frame(label_html, bbox)
    if not js_reply:
        break

    # convert JS response to OpenCV Image
    frame = js_to_image(js_reply["img"])

    # create transparent overlay for bounding box
    bbox_array = np.zeros([480,640,4], dtype=np.uint8)

    # call our darknet helper on video frame
    detections, width_ratio, height_ratio = darknet_helper(frame, width, height)

    # loop through detections and draw them on transparent overlay image
    for label, confidence, bbox in detections:
      left, top, right, bottom = bbox2points(bbox)
      left, top, right, bottom = int(left * width_ratio), int(top * height_ratio), int(right * width_ratio), int(bottom * height_ratio)
      bbox_array = cv2.rectangle(bbox_array, (left, top), (right, bottom), class_colors[label], 4)
      bbox_array = cv2.putText(bbox_array, "{} [{:.2f}]".format(label, float(confidence)),
                        (left, top - 5), cv2.FONT_HERSHEY_SIMPLEX, 0.5,
                        class_colors[label], 2)

    bbox_array[:,:,3] = (bbox_array.max(axis = 2) > 0 ).astype(int) * 255
    # convert overlay of bbox into bytes
    bbox_bytes = bbox_to_bytes(bbox_array)
    # update bbox so next frame gets new overlay
    bbox = bbox_bytes
```

Double Click Video than Open video

Video.mp4