**YOLO detects the person in a given image**

First, it divides the image into a 13×13 grid of cells. The size of these 169 cells vary depending on the size of the input. For a 416×416 input size that we used in our experiments, the cell size was 32×32. Each cell is then responsible for predicting a number of boxes in the image.

For each bounding box, the network also predicts the confidence that the bounding box actually encloses an object, and the probability of the enclosed object being a particular class.

Most of these bounding boxes are eliminated because their confidence is low or because they are enclosing the same object as another bounding box with very high confidence score. This technique is called **non-maximum suppression**.

## **Person Detection using YOLOv3**

### Step 1: Download the models

### The **yolov3.weights** file (containing the pre-trained network’s weights)

### The **yolov3.cfg** file (containing the network configuration)

### The **coco.names** file which contains the 80 different class names used in the COCO dataset

### Step 2: Initialize the parameters

The YOLOv3 algorithm generates bounding boxes as the predicted detection outputs. Every predicted box is associated with a confidence score. In the first stage, all the boxes below the confidence threshold parameter are ignored for further processing.

The rest of the boxes undergo **non-maximum suppression** which removes redundant overlapping bounding boxes. Non-maximum suppression is controlled by a parameter **nmsThreshold**. You can try to change these values and see how the number of output predicted boxes changes.

Next, the default values for the input width (**inpWidth**) and height (**inpHeight**) for the network’s input image are set. We set each of them to 416, so that we can compare our runs to the Darknet’s C code given by YOLOv3’s authors. You can also change both of them to 320 to get faster results or to 608 to get more accurate results.

### Step 3: Load the model and classes

The file coco.names contains all the objects for which the model was trained. We read class names.

Next, we load the network which has two parts —

1. yolov3.weights : The pre-trained weights.
2. yolov3.cfg : The configuration file.

We set the DNN backend to OpenCV here and the target to CPU. You could try setting the preferable target to cv.dnn.DNN\_TARGET\_OPENCL to run it on a GPU. But keep in mind that the current OpenCV version is tested only with Intel’s GPUs, it would automatically switch to CPU, if you do not have an Intel GPU.

### Step 4: Read the input

In this step we read the image, video stream or the webcam. In addition, we also open the video writer to save the frames with detected output bounding boxes.

### Step 5: Process each frame

The input image to a neural network needs to be in a certain format called a **blob**.

After a frame is read from the input image or video stream, it is passed through the **blobFromImage** function to convert it to an **input blob** for the neural network. In this process, it scales the image pixel values to a target range of 0 to 1 using a scale factor of 1/255. It also resizes the image to the given size of (416, 416) without cropping. Note that we do not perform any mean subtraction here, hence pass [0,0,0] to the mean parameter of the function and keep the swapRB parameter to its default value of 1.

The output blob is then passed in to the network as its input and a forward pass is run to get a list of predicted bounding boxes as the network’s output. These boxes go through a post-processing step in order to filter out the ones with low confidence scores. We will go through the post-processing step in more detail in the next section. We print out the inference time for each frame at the top left. The image with the final bounding boxes is then saved to the disk, either as an image for an image input or using a video writer for the input video stream.

#### **Step 6: Getting the names of output layers**

The **forward** function in OpenCV’s Net class needs the ending layer till which it should run in the network. Since we want to run through the whole network, we need to identify the last layer of the network. We do that by using the function **getUnconnectedOutLayers()** that gives the names of the unconnected output layers, which are essentially the last layers of the network. Then we run the forward pass of the network to get output from the output layers, as in the previous code snippet (**net.forward(getOutputsNames(net))**).

#### **Step 7: Post-processing the network’s output**

The network outputs bounding boxes are each represented by a vector of number of classes + 5 elements.

The first 4 elements represent the **center\_x**, **center\_y**, **width** and **height**. The fifth element represents the confidence that the bounding box encloses an object.

The rest of the elements are the confidence associated with each class (i.e. object type(person)). The box is assigned to the class corresponding to the highest score for the box.

The highest score for a box is also called its **confidence**. If the confidence of a box is less than the given threshold, the bounding box is dropped and not considered for further processing.

The boxes with their confidence equal to or greater than the confidence threshold are then subjected to **Non Maximum Suppression**. This would reduce the number of overlapping boxes.

#### **Step 8: Draw the predicted boxes**

Finally, we draw the boxes that were filtered through the non-maximum suppression, on the input frame with their assigned class label and confidence scores.