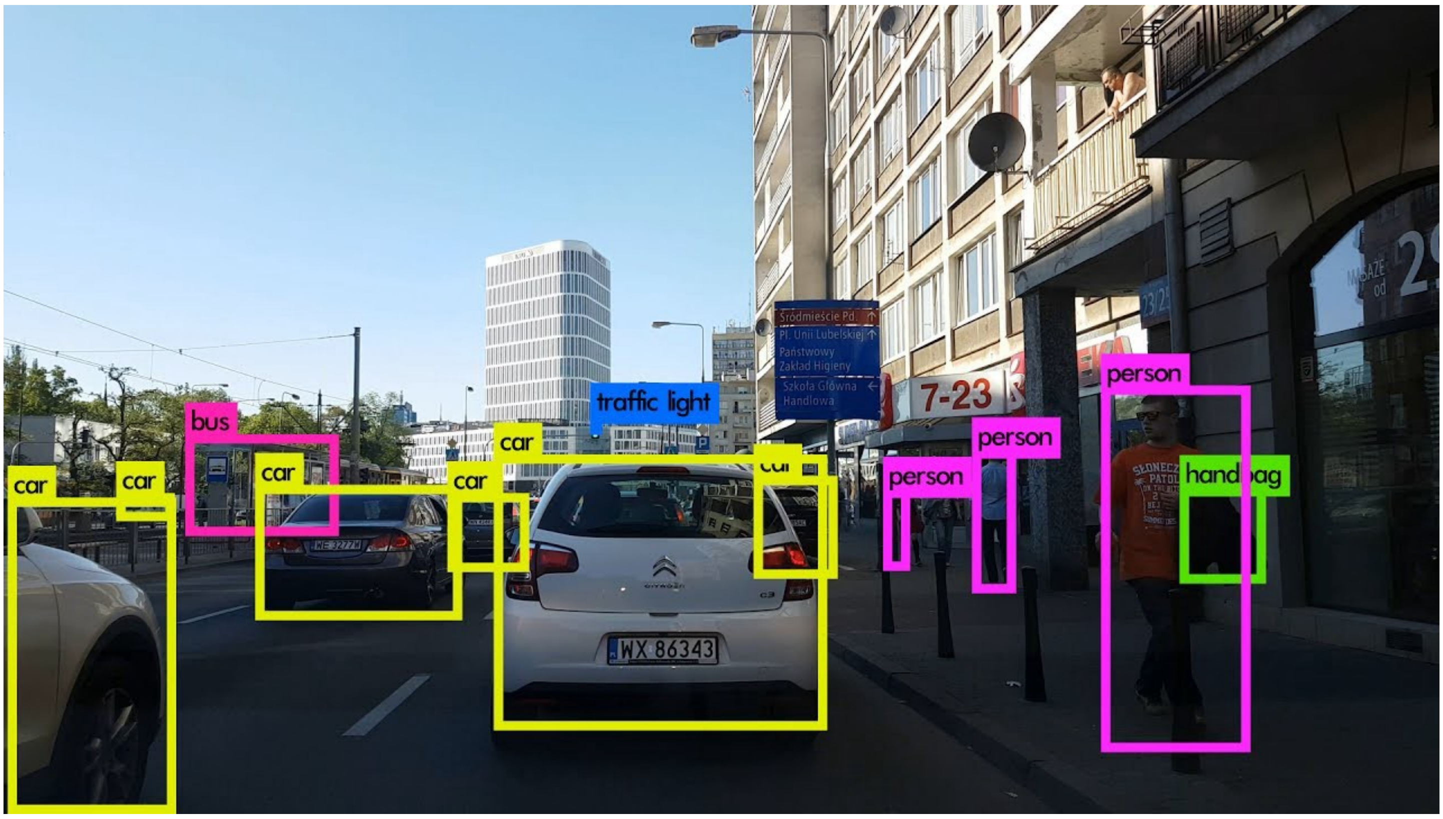
Ayoosh Kathuria Follow
Apr 23, 2018 · 9 min read



You only look once, or YOLO, is one of the faster object detection algorithms out there. Though it is no longer the most accurate object detection algorithm, it is a very good choice when you need real-time detection, without loss of too much accuracy.

A few weeks back, the third version of YOLO came out, and this post aims at explaining the changes introduced in YOLO v3. This is not going to be a post explaining what YOLO is from the ground up. I assume you know how YOLO v2 works. If that is not the case, I recommend you to check out the following papers by Joseph Redmon *et all*, to get a hang of how YOLO works.

1. YOLO v1
2. YOLO v2
3. A nice blog post on YOLO

YOLO v3: Better, not Faster, Stronger

The official title of YOLO v2 paper seemed if YOLO was a milk-based health drink for kids rather than a object detection algorithm. It was named “YOLO9000: Better, Faster, Stronger”.

For it’s time YOLO 9000 was the fastest, and also one of the most accurate algorithm. However, a couple of years down the line and it’s no longer the most accurate with algorithms like RetinaNet, and SSD outperforming it in terms of accuracy. It still, however, was one of the fastest.

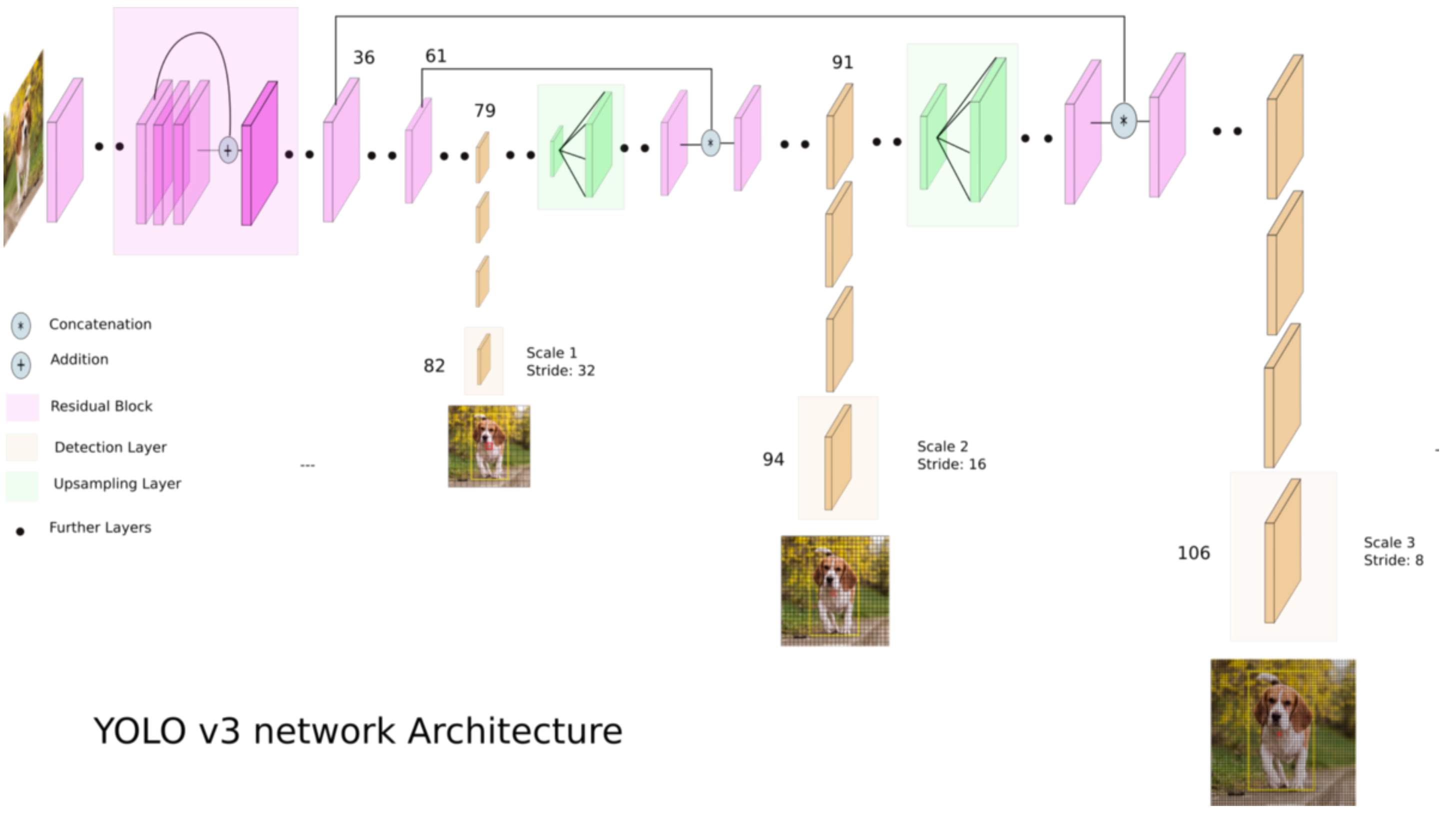
But that speed has been traded off for boosts in accuracy in YOLO v3. While the earlier variant ran on 45 FPS on a Titan X, the current version clocks about 30 FPS. This has to do with the increase in **complexity of underlying architecture called Darknet.**

Darknet-53

YOLO v2 used a custom deep architecture darknet-19, an originally 19-layer network supplemented with 11 more layers for object detection. With a 30-layer architecture, YOLO v2 often struggled with small object detections. This was attributed to loss of fine-grained features as the layers downsampled the input. To remedy this, YOLO v2 used an identity mapping, concatenating feature maps from from a previous layer to capture low level features.

However, YOLO v2’s architecture was still lacking some of the most important elements that are now staple in most of state-of-the art algorithms. No residual blocks, no skip connections and no upsampling. YOLO v3 incorporates all of these.

First, YOLO v3 uses a variant of Darknet, which originally has 53 layer network trained on Imagenet. For the task of detection, 53 more layers are stacked onto it, giving us a **106 layer fully convolutional underlying architecture for YOLO v3**. This is the reason behind the slowness of YOLO v3 compared to YOLO v2. Here is how the architecture of YOLO now looks like.



Ok, this diagram took a lot of time to make. I want a clap or ten for this!

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 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.**

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.

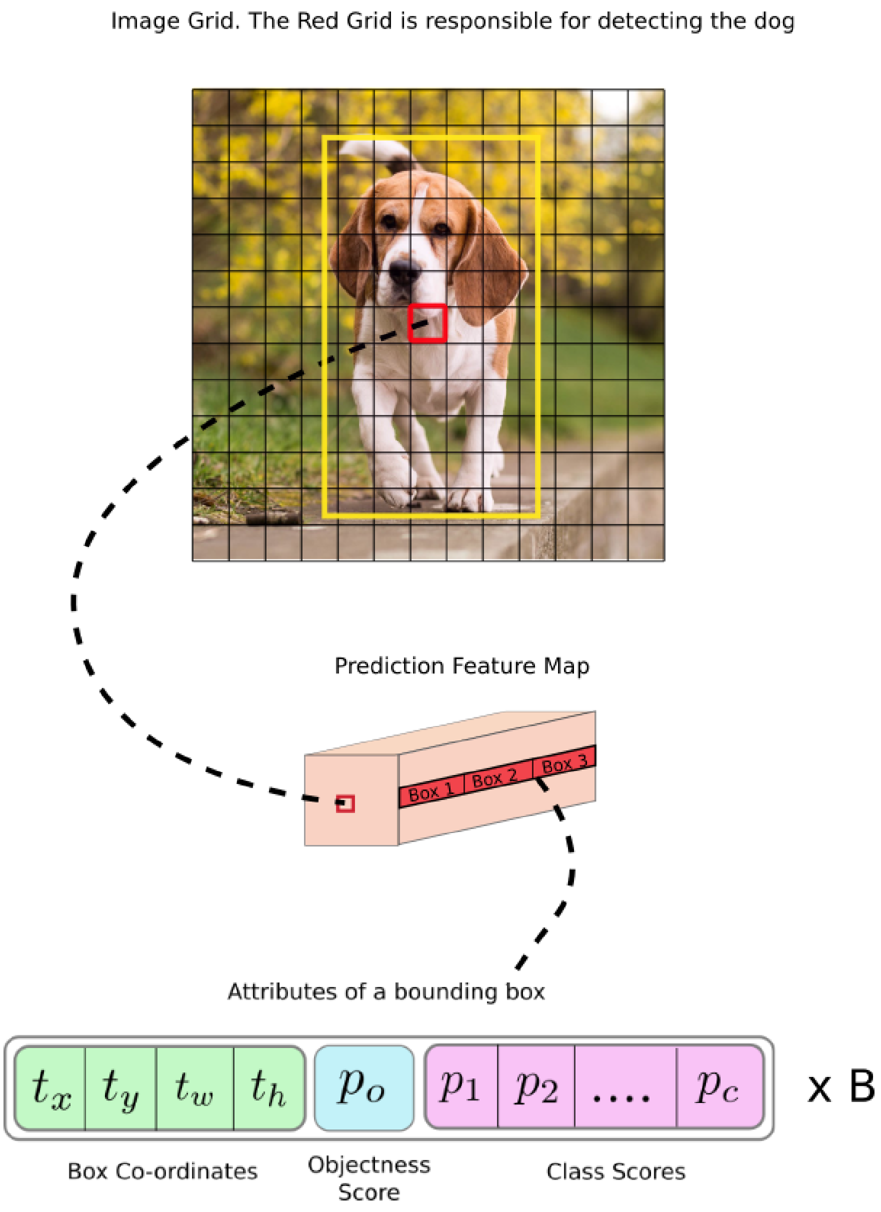


Image credits: https://blog.paperspace.com/how-to-implement-a-yolo-object-detector-in-pytorch/

Before we go further, I'd like to point out that **stride of the network, or a layer is defined as the ratio by which it downsamples the input.** In the following examples, I will assume we have an input image of size 416 x 416.

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.

Changes in Loss Function

Earlier, YOLO v2’s loss function looked like this.

$$
\begin{aligned}
&\lambda_{\textbf{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} (x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \\
&+ \lambda_{\textbf{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(\sqrt{w_i} - \sqrt{\hat{w}_i}\right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i}\right)^2 \\
&+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(C_i - \hat{C}_i\right)^2 \\
&+ \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i\right)^2 \\
&+ \sum_{i=0}^{S^2} \mathbb{1}_{i}^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2 \qquad (3)
\end{aligned}
$$

Image credits: https://pjreddie.com/media/files/papers/yolo_1.pdf

I know this is intimidating, but notice the last three terms. Of them, the first one penalizes the objectness score prediction for bounding boxes responsible for predicting objects (the scores for these should ideally be 1), the second one for bounding boxes having no objects, (the scores should ideally be zero), and the last one penalises the class prediction for the bounding box which predicts the objects.

The last three terms in YOLO v2 are the squared errors, whereas in YOLO v3, they’ve been replaced by cross-entropy error terms. In other words, **object confidence and class predictions in YOLO v3 are now predicted through logistic regression.**

While we are training the detector, for each ground truth box, we assign a bounding box, whose anchor has the maximum overlap with the ground truth box.

No more softmaxing the classes

YOLO v3 now performs multilabel classification for objects detected in images.

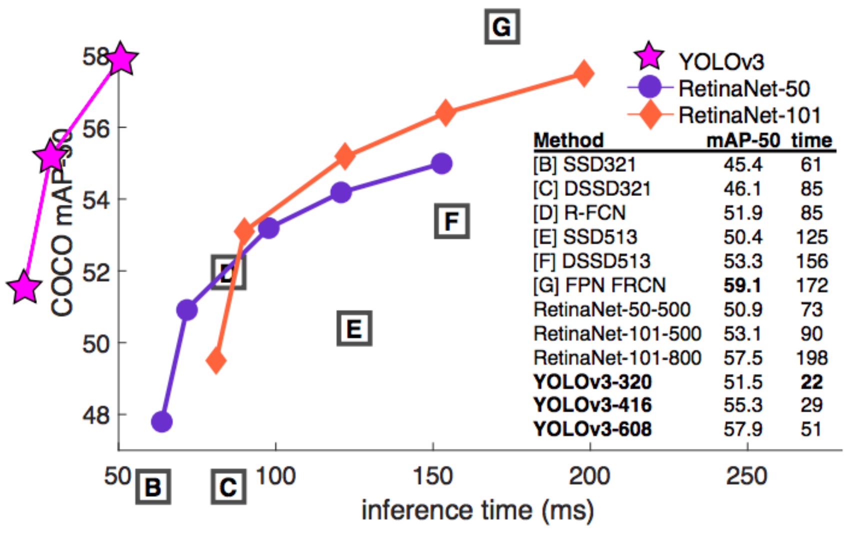
Earlier in YOLO, authors used to softmax the class scores and take the class with maximum score to be the class of the object contained in the bounding box. This has been modified in YOLO v3.

Softmaxing classes rests on the assumption that classes are mutually exclusive, or in simple words, if an object belongs to one class, then it cannot belong to the other. This works fine in COCO dataset.

However, when we have classes like **Person** and **Women** in a dataset, then the above assumption fails. This is the reason why the authors of YOLO have refrained from softmaxing the classes. Instead, each class score is predicted using logistic regression and a threshold is used to predict multiple labels for an object. Classes with scores higher than this threshold are assigned to the box.

Benchmarking

YOLO v3 performs at par with other state of art detectors like RetinaNet, while being considerably faster, at **COCO mAP 50 benchmark.** It is also better than SSD and it's variants. Here's a comparison of performances right from the paper.



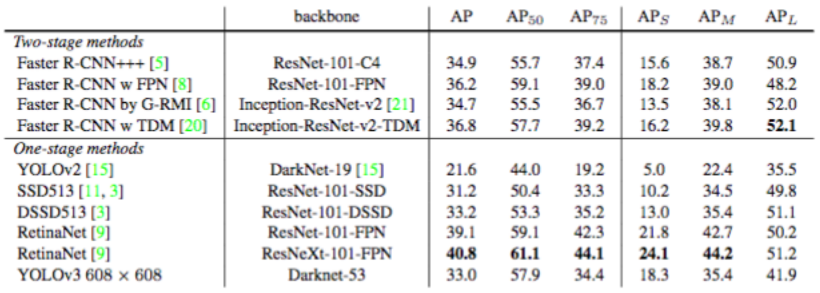
Method	mAP-50	time
[B] SSD321	45.4	61
[C] DSSD321	46.1	85
[D] R-FCN	51.9	85
[E] SSD513	50.4	125
[F] DSSD513	53.3	156
[G] FPN FRCN	**59.1**	172
RetinaNet-50-500	50.9	73
RetinaNet-101-500	53.1	90
RetinaNet-101-800	57.5	198
YOLOv3-320	51.5	**22**
YOLOv3-416	55.3	29
YOLOv3-608	57.9	51

YOLO vs RetinaNet performance on COCO 50 Benchmark

But, but and but, YOLO looses out on COCO benchmarks with a higher value of IoU used to reject a detection. I'm not going to explain how the

COCO benchmark works as it’s beyond the scope of the work, but the **50** in COCO 50 benchmark is a measure of how well do the predicted bounding boxes align the the ground truth boxes of the object. 50 here corresponds to 0.5 IoU. If the IoU between the prediction and the ground truth box is less than 0.5, the prediction is classified as a mislocalisation and marked a false positive.

In benchmarks, where this number is higher (say, COCO 75), the boxes need to be aligned more perfectly to be not rejected by the evaluation metric. Here is where YOLO is outdone by RetinaNet, as it’s bounding boxes are not aligned as well as of RetinaNet. Here’s a detailed table for a wider variety of benchmarks.



	backbone	AP	AP_{50}	AP_{75}	AP_S	AP_M	AP_L
Two-stage methods							
Faster R-CNN+++ [5]	ResNet-101-C4	34.9	55.7	37.4	15.6	38.7	50.9
Faster R-CNN w FPN [8]	ResNet-101-FPN	36.2	59.1	39.0	18.2	39.0	48.2
Faster R-CNN by G-RMI [6]	Inception-ResNet-v2 [21]	34.7	55.5	36.7	13.5	38.1	52.0
Faster R-CNN w TDM [20]	Inception-ResNet-v2-TDM	36.8	57.7	39.2	16.2	39.8	**52.1**
One-stage methods							
YOLOv2 [15]	DarkNet-19 [15]	21.6	44.0	19.2	5.0	22.4	35.5
SSD513 [11, 3]	ResNet-101-SSD	31.2	50.4	33.3	10.2	34.5	49.8
DSSD513 [3]	ResNet-101-DSSD	33.2	53.3	35.2	13.0	35.4	51.1
RetinaNet [9]	ResNet-101-FPN	39.1	59.1	42.3	21.8	42.7	50.2
RetinaNet [9]	ResNeXt-101-FPN	**40.8**	**61.1**	**44.1**	**24.1**	**44.2**	51.2
YOLOv3 608 × 608	Darknet-53	33.0	57.9	34.4	18.3	35.4	41.9

RetinaNet outperforms YOLO at COCO 75 benchmark. Notice, how RetinaNet is still better at the AP benchmark for small objects (APs)

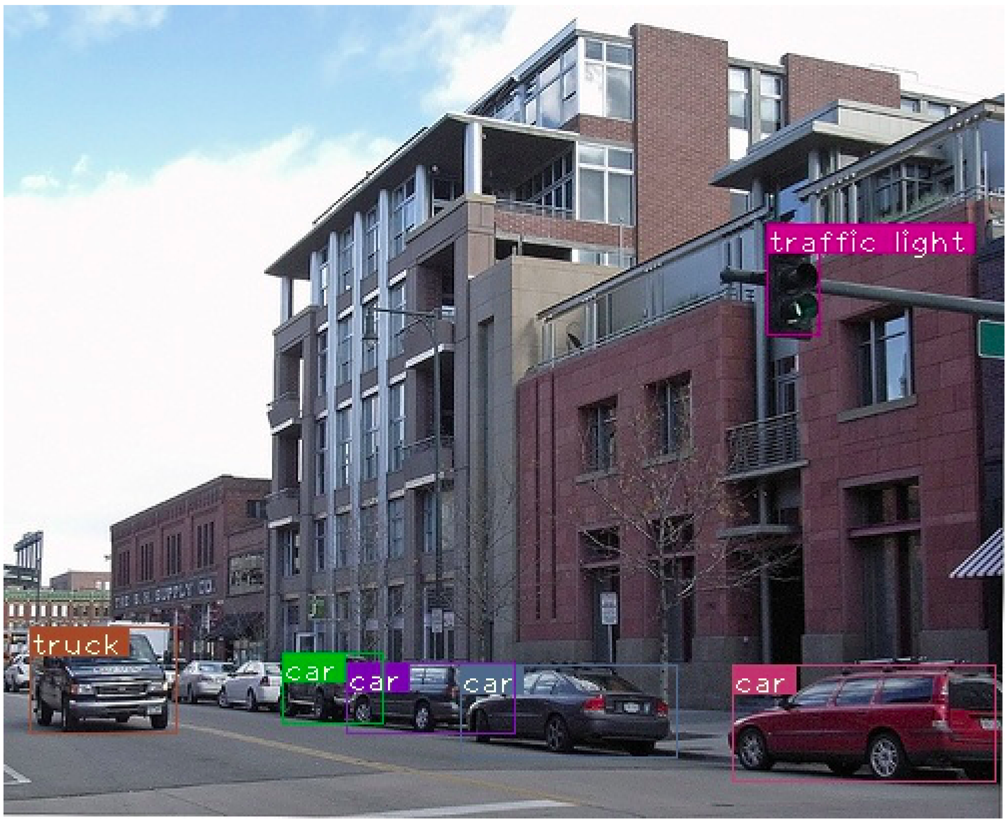
Doing some experiments

You can run the detector on either images or video by using the code provided in this Github repo. The code requires PyTorch 0.3+, OpenCV 3 and Python 3.5. Setup the repo, and you can run various experiments on it.

Different Scales

Here is a look at what the different detection layers pick up.

```
python detect.py --scales 1 --images imgs/img3.jpg
```

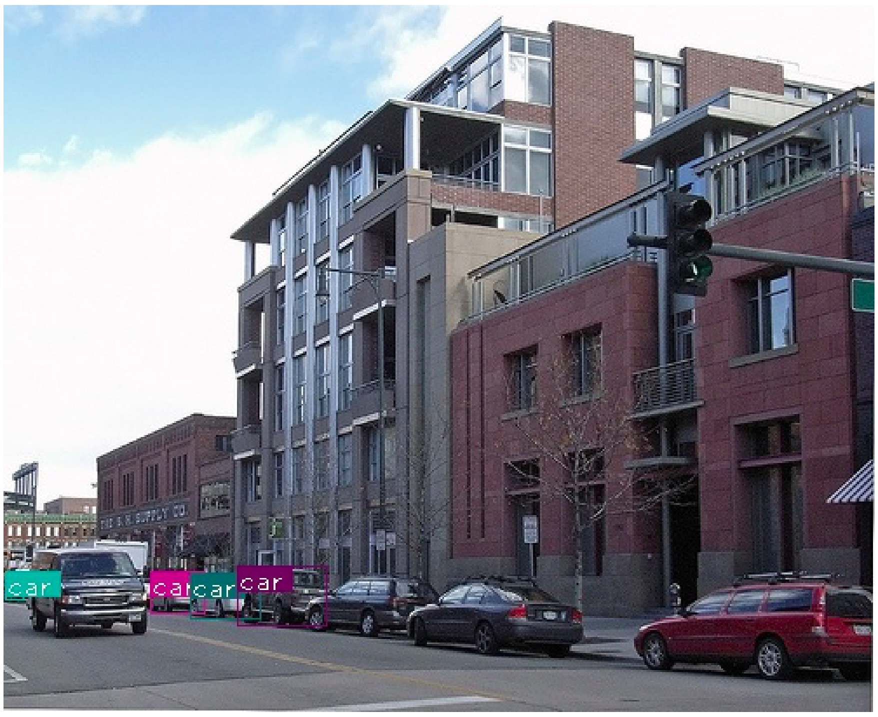


Detection at scale 1, we see somewhat large objects are picked. But we don’t detect a few cars.

```
python detect.py --scales 2 --images imgs/img3.jpg
```

No detections at scale 2.

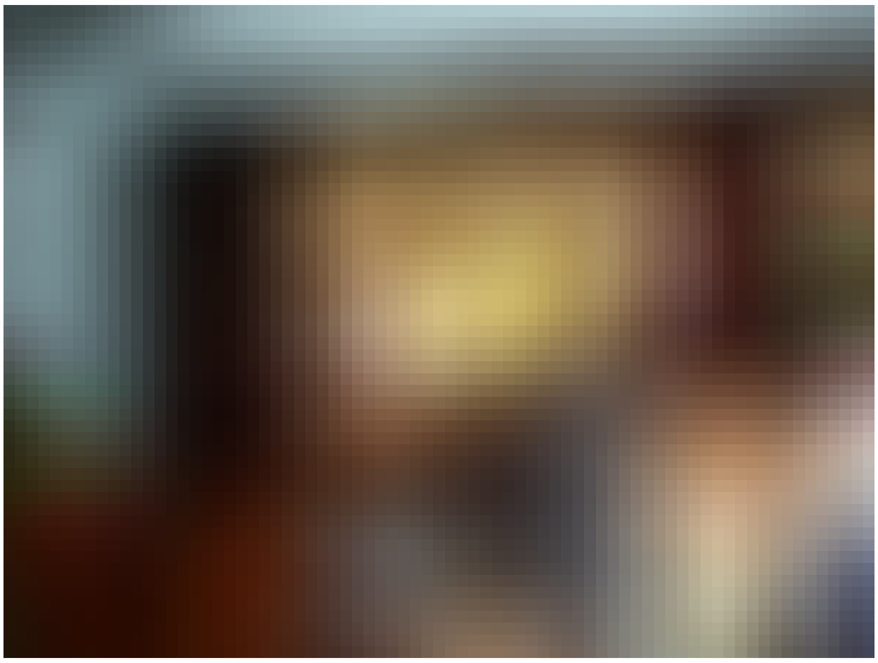
```
python detect.py --scales 3 --images imgs/img3.jpg
```



Detection at the largest scale (3). Look how only the small objects are picked up, which weren’t detected by scale 1.

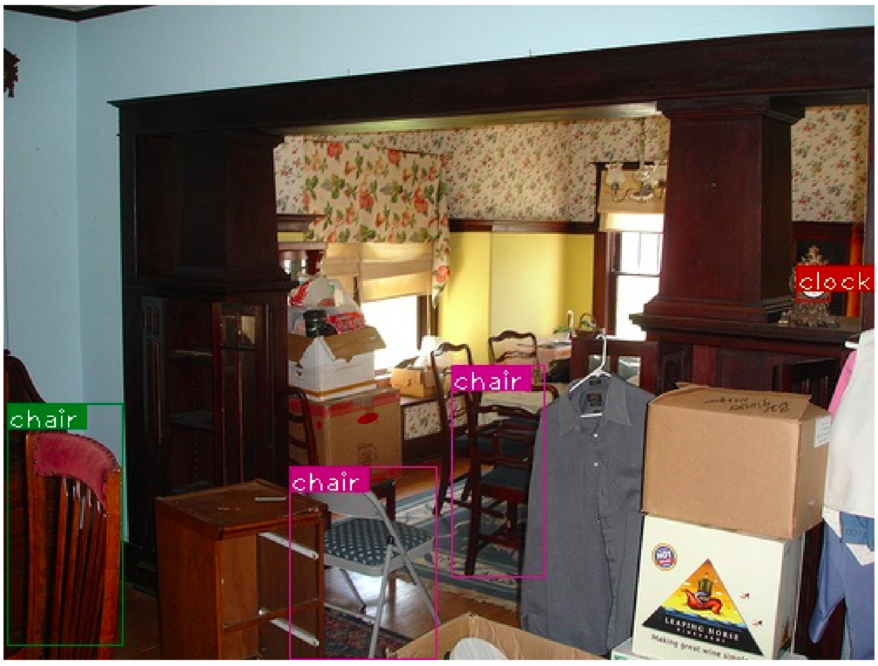
Different Input resolution

```
python detect.py --reso 320 --images imgs/imgs4.jpg
```



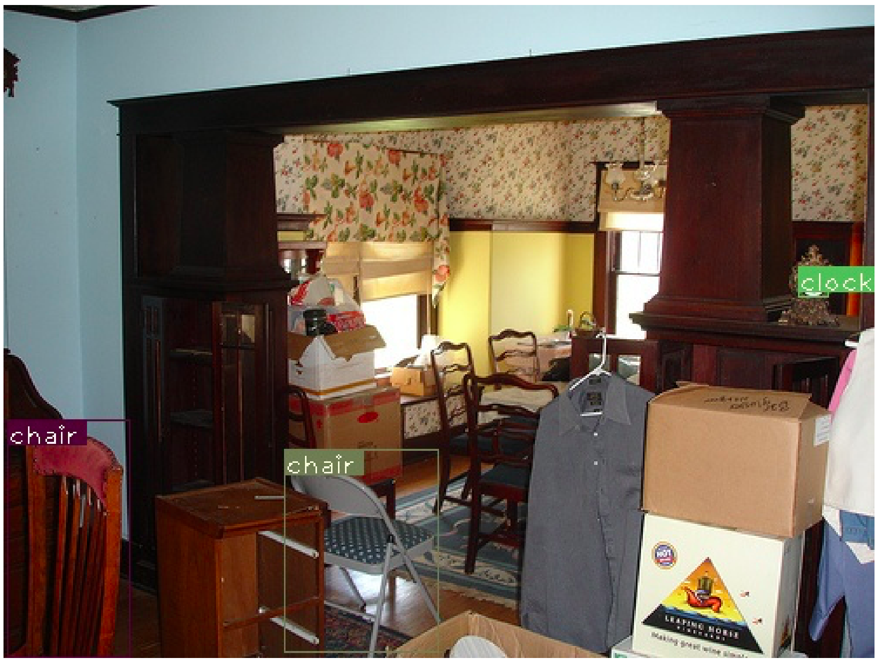
Input resolution of the image: 320 x 320

```
python detect.py --reso 416 --images imgs/imgs4.jpg
```



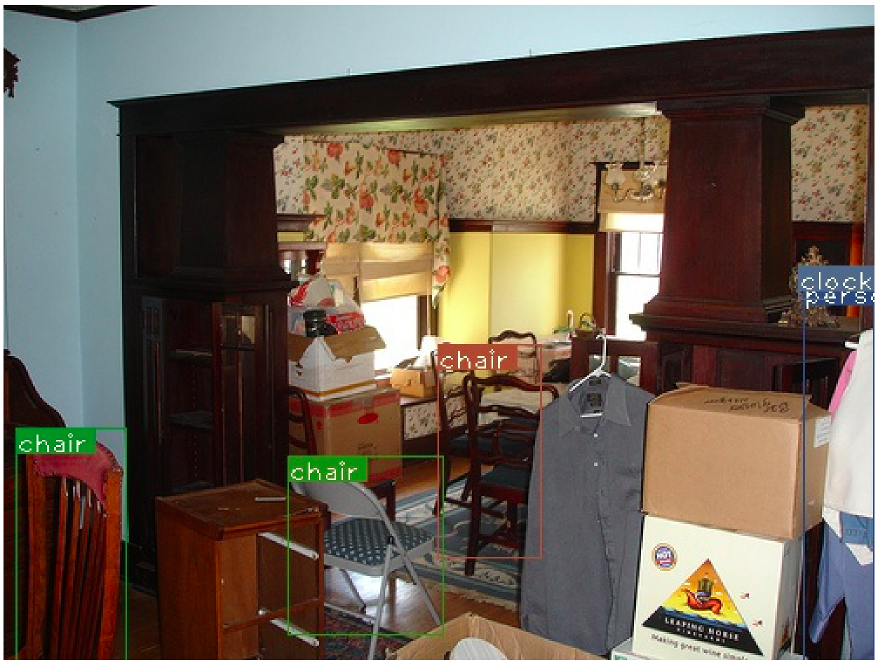
Input resolution of the image: 416 x 416

```
python detect.py --reso 608 --images imgs/imgs4.jpg
```



Here, we detect one less chair than before

```
python detect.py --reso 960 --images imgs/imgs4.jpg
```



Here, the detector picks up a false detection, the ”Person” at the right

Larger input resolutions don’t help much in our case, but they might help in detection of images with small objects. On the other hand, larger input resolutions add to inference time. This is a hyper parameter that needs to be tuned depending upon application.

You can experiment with other metrics such as batch size, objectness confidence, and NMS threshold by going to the repo. Everything has been mentioned in the ReadMe file.

Implementing YOLO v3 from scratch

If you want to implement a YOLO v3 detector by yourself in PyTorch, here's a series of tutorials I wrote to do the same over at Paperspace. Now, I'd expect you to have basic familiarity with PyTorch if you wanna have a go at this tutorial. If you're someone who's looking from moving from a beginner PyTorch user to an intermediate one, this tutorial is just about right.

Further Reading

1. YOLO v3: An incremental improvement
2. How is mAP calculated?