SERIES: YOLO OBJECT DETECTOR IN PYTORCH

How to implement a YOLO (v3) object detector from scratch in PyTorch: Part 1

Tutorial on building YOLO v3 detector from scratch detailing how to create the network architecture from a configuration file, load the weights and designing input/output pipelines.

Object detection is a domain that has benefited immensely from the recent developments in deep learning. Recent years have seen people develop many algorithms for object detection, some of which include YOLO, SSD, Mask RCNN and RetinaNet.

For the past few months, I've been working on improving object detection at a research lab. One of the biggest takeaways from this experience has been realizing that the best way to go about learning object detection is to implement the algorithms by yourself, from scratch. This is exactly what we'll do in this tutorial.

We will use PyTorch to implement an object detector based on YOLO v3, one of the faster object detection algorithms out there.

The code for this tutorial is designed to run on Python 3.5, and PyTorch **0.4**. It can be found in it's entirety at this Github repo.

This tutorial is broken into 5 parts:

1. Part 1 (This one): Understanding How YOLO works
2. Part 2 : Creating the layers of the network architecture
3. Part 3 : Implementing the the forward pass of the network
4. Part 4 : Objectness score thresholding and Non-maximum suppression
5. Part 5 : Designing the input and the output pipelines

Prerequisites

- You should understand how convolutional neural networks work. This also includes knowledge of Residual Blocks, skip connections, and Upsampling.
- What is object detection, bounding box regression, IoU and non-maximum suppression.
- Basic PyTorch usage. You should be able to create simple neural networks with ease.

I've provided the link at the end of the post in case you fall short on any front.

What is YOLO?

YOLO stands for You Only Look Once. It's an object detector that uses features learned by a deep convolutional neural network to detect an object. Before we get out hands dirty with code, we must understand how YOLO works.

A Fully Convolutional Neural Network

YOLO makes use of only convolutional layers, making it a fully convolutional network (FCN). It has 75 convolutional layers, with skip connections and upsampling layers. No form of pooling is used, and a convolutional layer with stride 2 is used to downsample the feature maps. This helps in preventing loss of low-level features often attributed to pooling.

Being a FCN, YOLO is invariant to the size of the input image. However, in practice, we might want to stick to a constant input size due to various problems that only show their heads when we are implementing the algorithm.

A big one amongst these problems is that if we want to process our images in batches (images in batches can be processed in parallel by the GPU, leading to speed boosts), we need to have all images of fixed height and width. This is needed to concatenate multiple images into a large batch (concatenating many PyTorch tensors into one)

The network downsamples the image by a factor called the **stride** of the network. For example, if the stride of the network is 32, then an input image of size 416 x 416 will yield an output of size 13 x 13. Generally, ***stride* of any layer in the network is equal to the factor by which the output of the layer is smaller than the input image to the network.**

Interpreting the output

Typically, (as is the case for all object detectors) the features learned by the convolutional layers are passed onto a classifier/regressor which makes the detection

prediction (coordinates of the bounding boxes, the class label.. etc).

In YOLO, the prediction is done by using a convolutional layer which uses 1 x 1 convolutions.

Now, the first thing to notice is our **output is a feature map**. Since we have used 1 x 1 convolutions, the size of the prediction map is exactly the size of the feature map before it. In YOLO v3 (and it's descendants), the way you interpret this prediction map is that each cell can predict a fixed number of bounding boxes.

> *Though the technically correct term to describe a unit in the feature map would be a neuron, calling it a cell makes it more intuitive in our context.*

Depth-wise, we have (B x (5 + C)) entries in the feature map. B represents the number of bounding boxes each cell can predict. According to the paper, each of these B bounding boxes may specialize in detecting a certain kind of object. Each of the bounding boxes have $5 + C$ attributes, which describe the center coordinates, the dimensions, the objectness score and C class confidences for each bounding box. YOLO v3 predicts 3 bounding boxes for every cell.

You expect each cell of the feature map to predict an object through one of it's bounding boxes if the center of the object falls in the receptive field of that cell. (Receptive field is the region of the input image visible to the cell. Refer to the link on convolutional neural networks for further clarification).

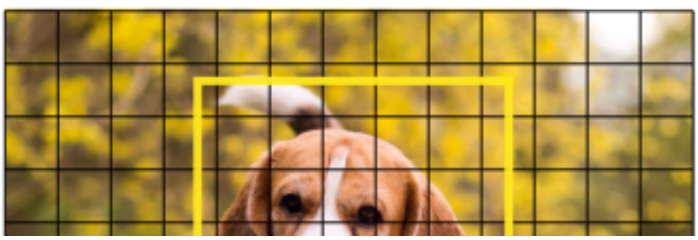
This has to do with how YOLO is trained, where only one bounding box is responsible

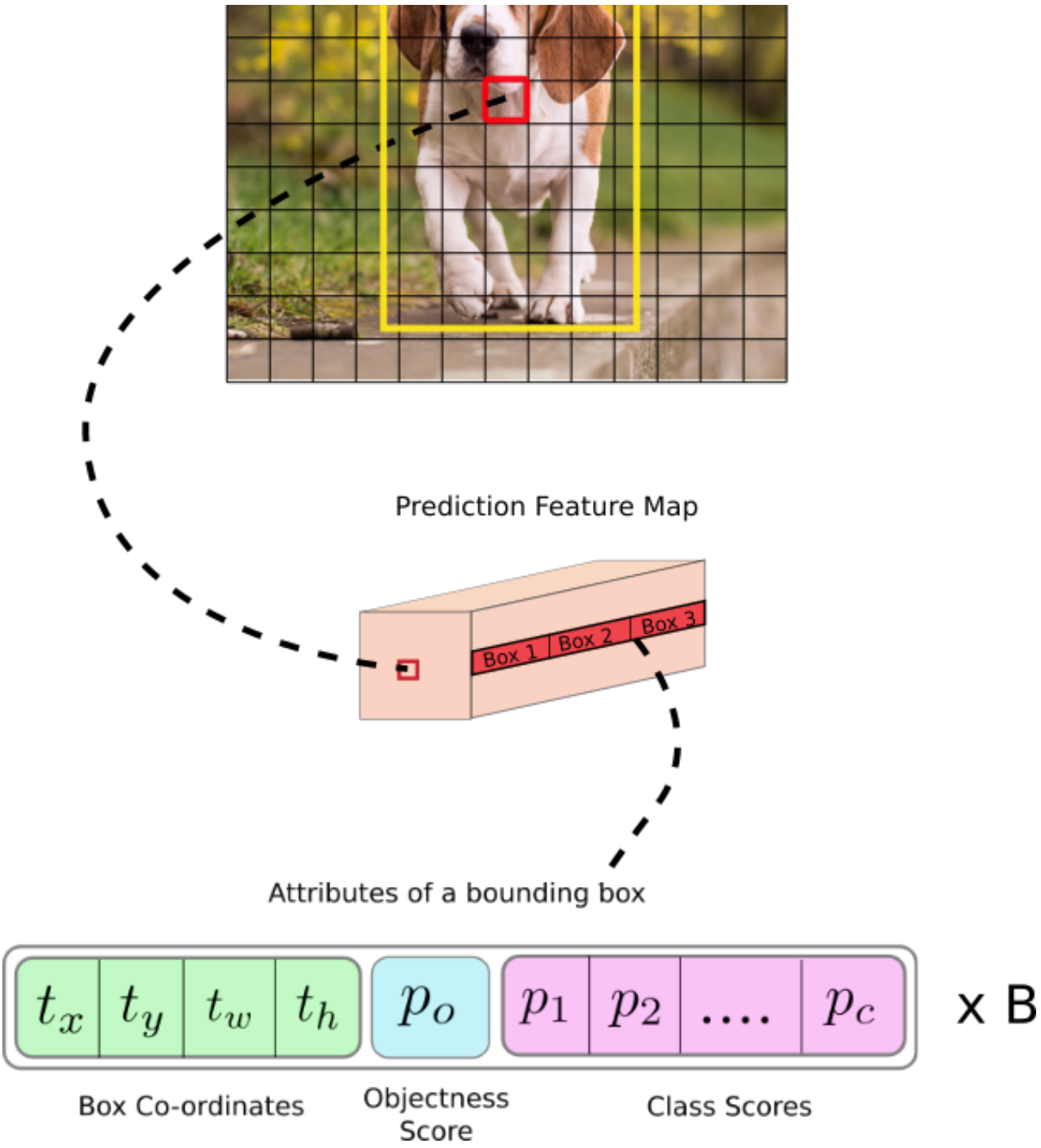
for detecting any given object. First, we must ascertain which of the cells this bounding box belongs to.

To do that, we divide the **input** image into a grid of dimensions equal to that of the final feature map.

Let us consider an example below, where the input image is 416 x 416, and stride of the network is 32. As pointed earlier, the dimensions of the feature map will be 13 x 13. We then divide the input image into 13 x 13 cells.

Image Grid. The Red Grid is responsible for detecting the dog





Then, the cell (*on the input image*) containing the center of the ground truth box of an object is chosen to be the one responsible for predicting the object. In the image, it is

the cell which marked red, which contains the center of the ground truth box (marked yellow).

Now, the red cell is the 7th cell in the 7th row on the grid. We now assign the 7th cell in

the 7th row **on the feature map** (corresponding cell on the feature map) as the one responsible for detecting the dog.

Now, this cell can predict three bounding boxes. Which one will be assigned to the dog's ground truth label? In order to understand that, we must wrap out head around the concept of anchors.

> *Note that the cell we're talking about here is a cell on the prediction feature map. We divide the **input image** into a grid just to determine which cell of the **prediction feature map** is responsible for prediction*

Anchor Boxes

It might make sense to predict the width and the height of the bounding box, but in practice, that leads to unstable gradients during training. Instead, most of the modern object detectors predict log-space transforms, or simply offsets to pre-defined default bounding boxes called **anchors**.

Then, these transforms are applied to the anchor boxes to obtain the prediction. YOLO v3 has three anchors, which result in prediction of three bounding boxes per cell.

Coming back to our earlier question, the bounding box responsible for detecting the dog will be the one whose anchor has the highest IoU with the ground truth box.

Making Predictions

The following formulae describe how the network output is transformed to obtain

bounding box predictions.

$$b_x = \sigma(t_x) + c_x$$
$$b_y = \sigma(t_y) + c_y$$
$$b_w = p_w e^{t_w}$$
$$b_h = p_h e^{t_h}$$

bx, by, bw, bh are the x,y center co-ordinates, width and height of our prediction. *tx, ty, tw, th* is what the network outputs. *cx* and *cy* are the top-left co-ordinates of the grid. *pw* and *ph* are anchors dimensions for the box.

Center Coordinates

Notice we are running our center coordinates prediction through a sigmoid function. This forces the value of the output to be between 0 and 1. Why should this be the case? Bear with me.

Normally, YOLO doesn't predict the absolute coordinates of the bounding box's center.

It predicts offsets which are:

- Relative to the top left corner of the grid cell which is predicting the object.
- Normalised by the dimensions of the cell from the feature map, which is, 1.

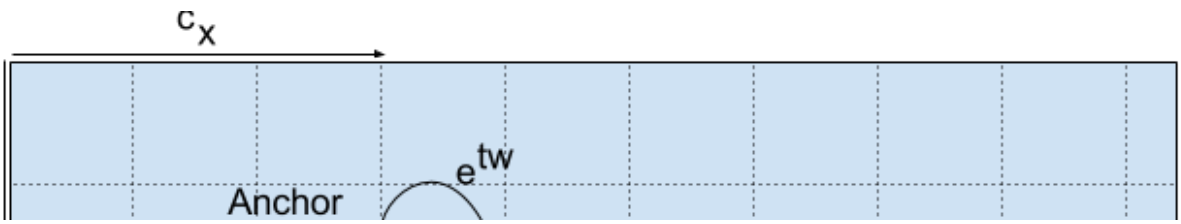
For example, consider the case of our dog image. If the prediction for center is (0.4, 0.7), then this means that the center lies at (6.4, 6.7) on the 13 x 13 feature map. (Since the top-left co-ordinates of the red cell are (6,6)).

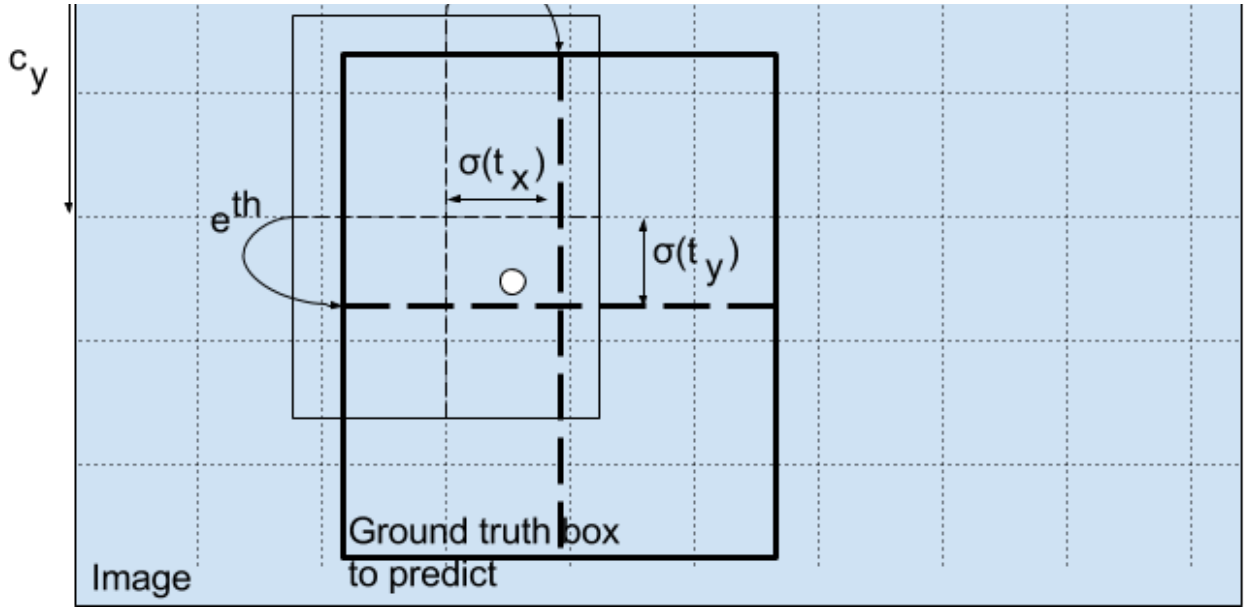
But wait, what happens if the predicted x,y co-ordinates are greater than one, say (1.2, 0.7). This means center lies at (7.2, 6.7). Notice the center now lies in cell just right to our red cell, or the 8th cell in the 7th row. **This breaks theory behind YOLO** because if we postulate that the red box is responsible for predicting the dog, the center of the dog must lie in the red cell, and not in the one beside it.

Therefore, to remedy this problem, the output is passed through a sigmoid function, which squashes the output in a range from 0 to 1, effectively keeping the center in the grid which is predicting.

Dimensions of the Bounding Box

The dimensions of the bounding box are predicted by applying a log-space transform to the output and then multiplying with an anchor.





How the detector output is transformed to give the final prediction. Image Credits. http://christopher5106.github.io/

The resultant predictions, *bw* and *bh*, are normalised by the height and width of the image. (Training labels are chosen this way). So, if the predictions *bx* and *by*for the box containing the dog are (0.3, 0.8), then the actual width and height on 13 x 13 feature map is (13 x 0.3, 13 x 0.8).

Objectness Score

Object score represents the probability that an object is contained inside a bounding box. It should be nearly 1 for the red and the neighboring grids, whereas almost 0 for, say, the grid at the corners.

The objectness score is also passed through a sigmoid, as it is to be interpreted as a probability.

Class Confidences

Class confidences represent the probabilities of the detected object belonging to a particular class (Dog, cat, banana, car etc). Before v3, YOLO used to softmax the class scores.

However, that design choice has been dropped in v3, and authors have opted for using sigmoid instead. The reason is that Softmaxing class scores assume that the classes are mutually exclusive. In simple words, if an object belongs to one class, then it's guaranteed it cannot belong to another class. This is true for COCO database on which we will base our detector.

However, this assumptions may not hold when we have classes like *Women* and *Person*. This is the reason that authors have steered clear of using a Softmax activation.

Prediction across different scales.

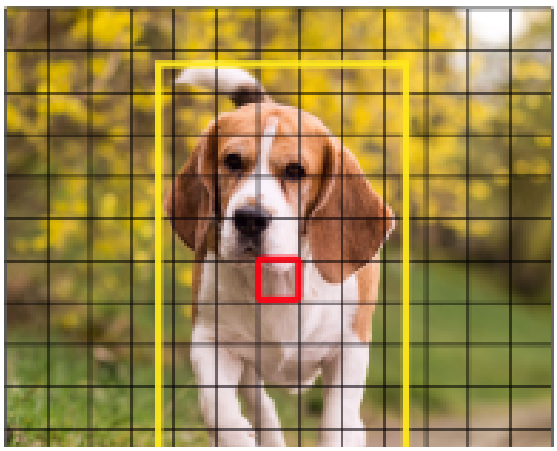
YOLO v3 makes prediction across 3 different scales. The detection layer is used make detection at feature maps of three different sizes, having **strides 32, 16, 8** respectively. This means, with an input of 416 x 416, we make detections on scales 13 x 13, 26 x 26 and 52 x 52.

The network downsamples the input image until the first detection layer, where a detection is made using feature maps of a layer with stride 32. Further, layers are upsampled by a factor of 2 and concatenated with feature maps of a previous layers having identical feature map sizes. Another detection is now made at layer with stride

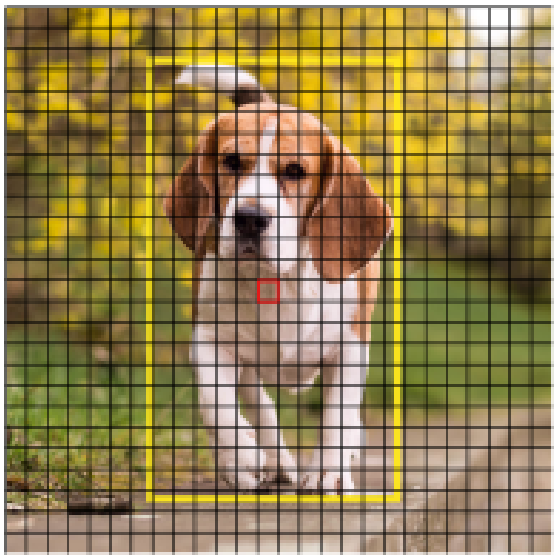
having identical feature map sizes. Another detection is now made at layer with stride 16. The same upsampling procedure is repeated, and a final detection is made at the layer of stride 8.

At each scale, each cell predicts 3 bounding boxes using 3 anchors, making the total number of anchors used 9. (The anchors are different for different scales)

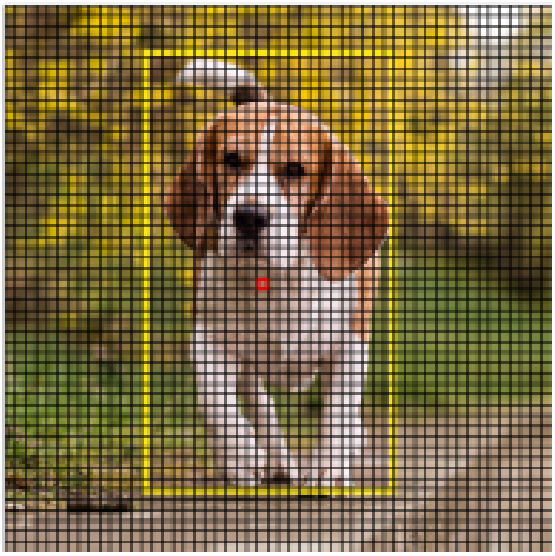
Prediction Feature Maps at different Scales



13 x 13



26 x 26



52 x 52

The authors report that this helps YOLO v3 get better at detecting small objects, a frequent complaint with the earlier versions of YOLO. Upsampling can help the network learn fine-grained features which are instrumental for detecting small objects.

Output Processing

For an image of size 416 x 416, YOLO predicts ((52 x 52) + (26 x 26) + 13 x 13)) x 3 = **10647 bounding boxes**. However, in case of our image, there's only one object, a

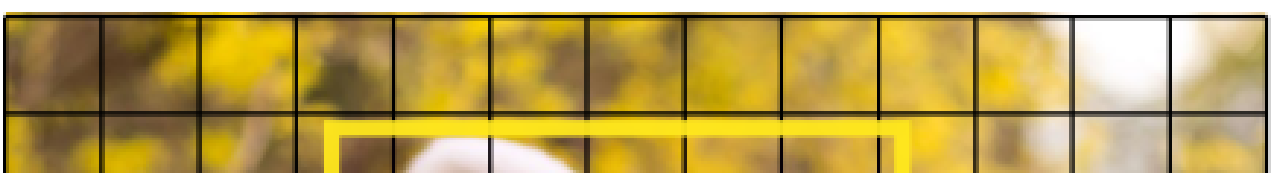
dog. How do we reduce the detections from 10647 to 1?

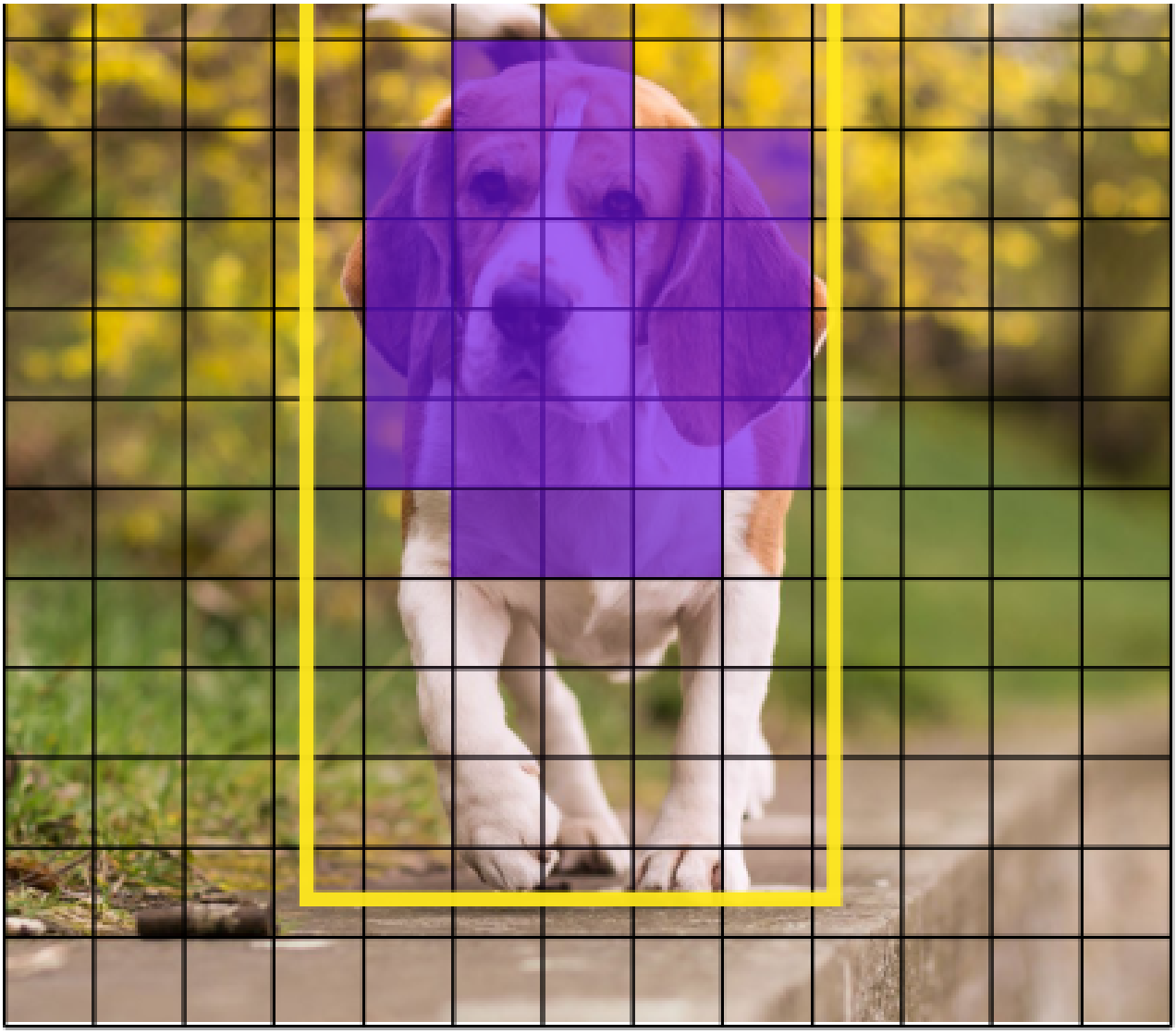
Thresholding by Object Confidence

First, we filter boxes based on their objectness score. Generally, boxes having scores below a threshold are ignored.

Non-maximum Suppression

NMS intends to cure the problem of multiple detections of the same image. For example, all the 3 bounding boxes of the red grid cell may detect a box or the adjacent cells may detect the same object.





Multiple Grids may detect the same object
NMS is used to remove multiple detections

If you don't know about NMS, I've provided a link to a website explaining the same.

Our Implementation

YOLO can only detect objects belonging to the classes present in the dataset used to train the network. We will be using the official weight file for our detector. These weights have been obtained by training the network on COCO dataset, and therefore we can detect 80 object categories.

That's it for the first part. This post explains enough about the YOLO algorithm to enable you to implement the detector. However, if you want to dig deep into how YOLO works, how it's trained and how it performs compared to other detectors, you can read the original papers, the links of which I've provided below.

That's it for this part. In the next part, we implement various layers required to put together the detector.

Further Reading

1. YOLO V1: You Only Look Once: Unified, Real-Time Object Detection
2. YOLO V2: YOLO9000: Better, Faster, Stronger
3. YOLO V3: An Incremental Improvement
4. Convolutional Neural Networks
5. Bounding Box Regression (Appendix C)
6. IoU
7. Non maximum suppresion
8. PyTorch Official Tutorial

Ayoosh Kathuria is currently an intern at the Defense Research and Development Organization, India, where he is working on improving object detection in grainy videos. When he's not working, he's either sleeping or playing pink floyd on his guitar. You can connect with him on LinkedIn or look at more of what he does at GitHub