YOLO v3 for Object Detection

contact: *cursifrancesco@gmail.com*

INTRODUCTION

YOLO V3: ARCHITECTURE

YOLO v3 is made of a sequence of convolutional layers with skip connections. Its architecture consists of different types of blocks:

- **Convolutional**: a traditional convolutional block consisting of convolutional layer with eventual batch normalization, but no pooling. It is defined by the **output channels**, **kernel size**, **stride length**, **padding**. When padding is used the pad dimension is set to $\frac{kernel\_size-1}{2}$. This makes sure that the input figure is not resized.
- **Shortcut**: it is a skip connection whose output is obtained by adding the output of the previous layer to a past feature map.
- **Upsample**: upsamples the feture map in the previous layer by using bilinear upsampling.
- **Route**: similar to the shortcut, it is used to output either feature maps from a specific layer, or to concatenate (along the depth dimension) feature maps from other layers.
- **Yolo**: it is the detection layer. It is defined by a **mask** to choose the indexes of the anchor boxes, **anchors** that define the width and height of the anchor boxes.

A full architecture diagram is shown in the figure

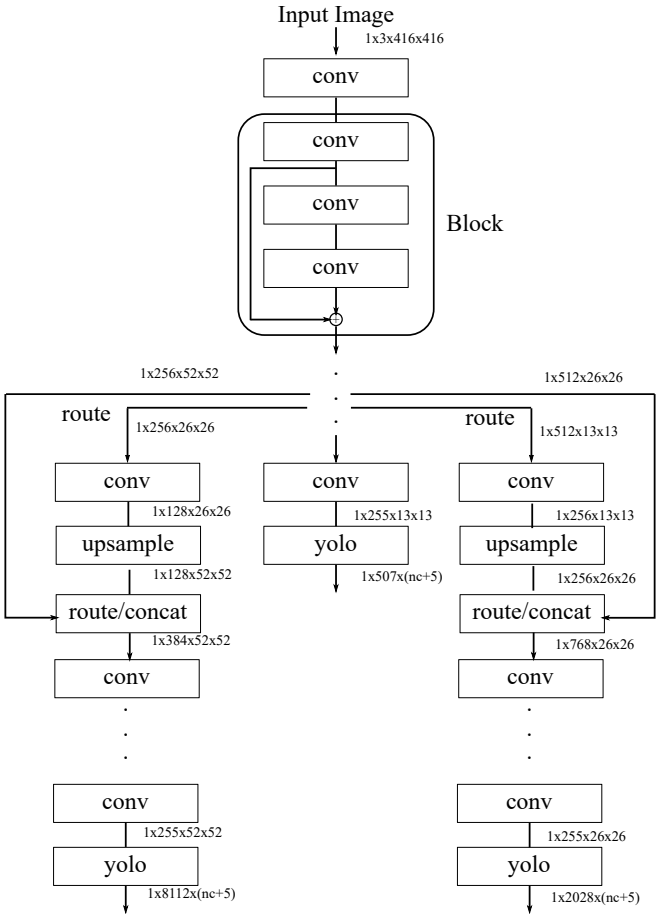

Fig. 1: Yolo v3 architecture and its layers

Network Output

The neural network output is obtained by combining the outputs of the three different extractors that act at different levels: at resolution of 13×13, 26×26, and 52×52 pixels. For each grid element of these features map, since 3 anchor boxes are

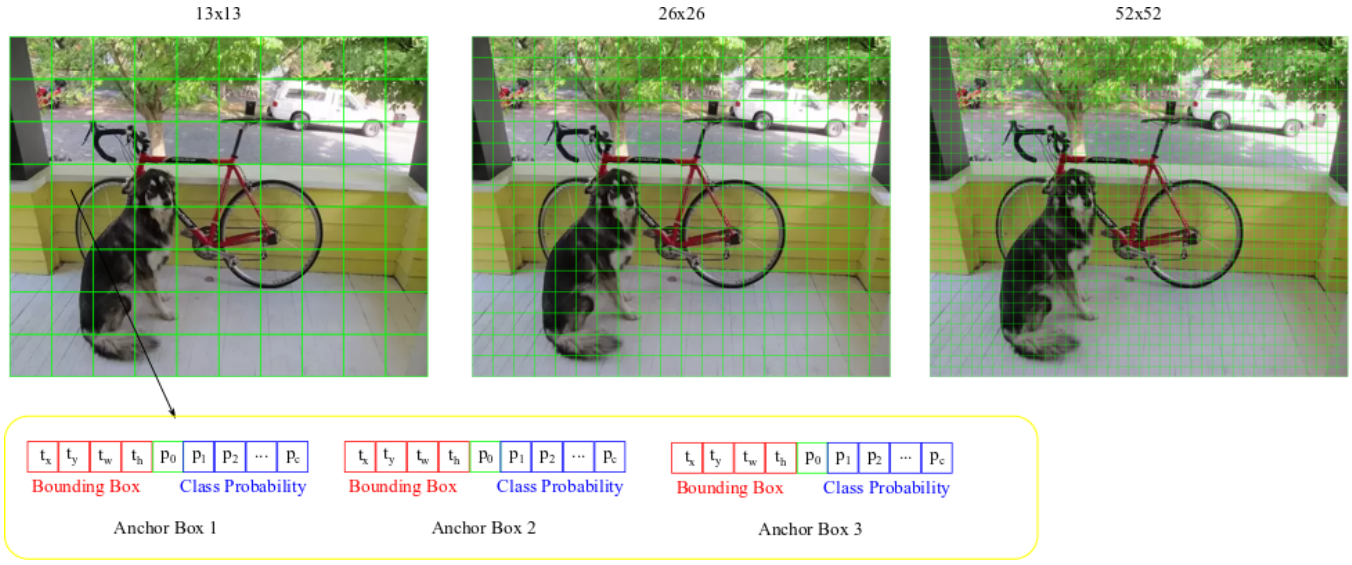

Fig. 2: YOLOv3 feature maps and outputs for each bounding box

used, the network then outputs a vector in $\mathbb{R}^{10647\times(n_c+5)}$, where n_c is the number of classes to identify. Additionally, the output dimension of the last convolutionl layer before the yolo layer is $3*(n_c+5)$. This is because each cell in the feature map tries to predict a n_c+5-dimensional vector for each anchor box. The additional 5 values correspond to the information about the bounding box t_x, t_y, t_w, t_h and objectness score p_0 (the probability of having an object in the cell).

Anchor Boxes are user pre-defined boxes dimensions (width and height) that are generally used to help the network identify objects in the scene.

The network final bounding box prediction is obtained as:

$$b_x = \sigma(t_x) + c_x, \quad b_y = \sigma(t_y) + c_y, \quad b_w = p_w e^{t_w}, \quad b_h = p_h e^{t_h} \ , \tag{1}$$

where b_x, b_y, b_w, b_h are the bounding box dimensions, c_x, c_y the coordinates of each cell of the grid, p_w, p_y the anchor box dimensions, and $\sigma(\cdot)$ the sigmoid function.

Non-Max Suppression and IoU

In order to reduce the number of elements to analyse and obtain a final output, different steps are taken.

First, bounding boxes with small confidence (small objectiveness score) are neglected. In this case, the prediction vector corresponding to low confidence is set to zero.

Different bounding boxes may identify the same object. **Non-Max Suppression** allows retaining only one bounding box with the highest confidence, among those detecting the same object. The algorithm takes into consideration the objectiveness score and the **Intersection over Unit (IoU)** measure. It works iteratively, for each class detected, as:

1) select bounding box with highest objectiveness score;
2) compare IoU of this box with the other bounding boxes detecting the same object;
3) remove the bounding boxes with an IoU (overlap) greater than a threshold. This means that they are not needed as the most confident box is already significative;
4) move to the next box with highest score;
5) repeat steps 2 to 4.

IoU measures the ration between the area of the intersection and the area of the union of two bounding boxes. The higher this value (the closer to 1), the more the boxes overlap. The smaller the value (closer to 0), the less they overlap.

The YOLO prediction will then be a $D\times 8$ matrix, where D are the number of bounding boxes detected in each image and each row corresponds to each bounding box location and size, objectiveness score, score of class with maximum confidence, and the class index.

YOLO v3: Loss Function

The loss to train a YOLO network consists of different terms, for each of the proposed anchor boxes and for each feature map:

- Binary Cross-Entropy **Objectiveness Loss**: it is used to make sure the network successfully identifies the locations in the feature maps that contain objects.
- **IoU Loss**: for the locations with objects (as from training data) the loss minimizes the IoU between the predicted and the actual bounding boxes.
- **Box Coordinates Loss**: minimizes the error between the coordinates of the predicted and the target bounding boxes.
- **Object Class Loss**: minimizes the loss between the predicted and the target object class.

Rearranging the Training Data: In order to properly compute the loss, the target training data need to be represented properly. For YOLO, the target data comes in the for of images and a *.txt* files. Each image has a corresponding *.txt* file containing the target bounding boxes coordinates and the corresponding class label. Each *.txt* file can, therefore, contain more than one line, depending on how many objects there are in the image. The bounding boxes' coordinates in the *.txt*

Fig. 3: Example of YOLO training data

file are expressed as percentages with respect to the image sizes (e.g. $x = 0.32$ means $x = 0.32W$, where W is the image width).

Depending on the chosen architecture of the YOLO network and its corresponding output, the training data need to be rearranged to simplify the computations. In this case, YOLO outputs 3 feature maps, arranged in a list of 3 elements, of dimensions $\hat{F}_i \in \mathbb{R}^{B \times f_{w,i} \times f_{h,i} \times (n_c+5) \times 3}$, where B is the batch size (number of images), $f_{w,i}, f_{h,i}$ the feature map dimensions (e.g. 13, 26, 52), n_c the number of classes in the data, and 3 is the number of anchor boxes.

In my case, the target data is therefore rearranged as a list of 3 tensors with sizes $T_i \in \mathbb{R}^{B \times f_{w,i} \times f_{h,i} \times (n_c+5)}$ for each feature map i.

Computing the Loss: Let's consider the output of one predicted feature map F_i and the corresponding training target T_i. First, the locations containing or not any objects need to be identified and the different losses initialized:

```
obj = target[...,4] == 1
noobj = target[..., 4] == 0
idx_obj = torch.argwhere(obj).squeeze(0)
idx_noobj = torch.argwhere(noobj).squeeze(0)

loss_noobj = 0.
loss_classes = 0.
loss_IoU = 0.
loss_coord = 0.
```

Then, for each anchor box, the losses are computed. For the locations that do not contain objects, the **Objectiveness Loss** is computed as:

```
loss_noobj = loss_noobj+BCEobj(pred_obj_score[idx_noobj[:,0],idx_noobj[:,1],idx_noobj[:,2]],
                    target[idx_noobj[:,0],idx_noobj[:,1],idx_noobj[:,2],4])
```

For the locations containing objects, instead, the corresponding predicted and target bounding boxes, and predicted and target class labels are obtained and the **IoU Loss**, **Box Coordinates Loss**, and **Object Class Loss** computed as:

```
#———Bounding box
pred_bbox = pred_bbox[idx_obj[:, 0], idx_obj[:, 1], idx_obj[:, 2],:]
```

```
            target_bbox = target_bbox[idx_obj[:, 0], idx_obj[:, 1], idx_obj[:, 2], :]

            pred_bbox[:,0:2] = torch.sigmoid(pred_bbox[:,0:2])+idx_obj[:, 1:]
            pred_bbox[:, 2:] = torch.exp(pred_bbox[:, 2:]) * anchor

            #————————— 2) Object Loss IoU
            IoU = bbox_iou(target_bbox,pred_bbox,xyxy=False)
            loss_IoU = loss_IoU+(1.0 - IoU).mean()

            #————————— 3) Object Loss, Box Cooridnates
            loss_coord = loss_coord+MSE_loss(pred_bbox,target_bbox)

            # ————————— 4) Object Loss, Classes loss
            target_classes = target[idx_obj[:,0],idx_obj[:,1],idx_obj[:,2],5:]
            pred_classes = pred_classes[idx_obj[:, 0], idx_obj[:, 1], idx_obj[:, 2],:]
            loss_classes = loss_classes+BCEcls(pred_classes, target_classes)
```

The overall loss for that anchor can then be computes as a weighted sum:

```
loss_anchor = loss_anchor+ 0.05*loss_IoU+0.05*loss_coord+1.0*loss_noobj+1*loss_classes
```

This computations need to be repeated for each anchor box and for each feature map, summing up all the losses.