Chapter 10

Object Detection – YOLO

In the ever-evolving landscape of deep learning, object detection stands out as a crucial task with wide-ranging applications, from autonomous vehicles to surveillance systems. In this chapter, we delve into the intricacies of one of the most influential and efficient object detection algorithms—You Only Look Once (YOLO). YOLO revolutionized the field by introducing a real-time, one-pass approach to detecting objects in images, streamlining the detection process significantly. This chapter explores the underlying principles of YOLO, detailing its unique architecture and the principles that make it stand apart. From its inception (YOLO v1) to a few improved versions (YOLO v2 and YOLO v3), we will guide you through the evolution of YOLO, providing insights into its strengths, limitations, and practical implementations.

After completing this chapter, one should be able to

- Understand the basic concepts of object detection in computer vision.
- Know the details about the architectures of YOLO versions (YOLO1, YOLO2, YOLO3)
- Understand the loss function of YOLO.
- Implement a YOLO model from scratch in PyTorch.
- Train a YOLO model.

10.1 Introduction

The task of image classification is to predict an image as one of the pre-defined categories, for instance, to classify the picture as “car” or “non-car”. The task of object detection is to deal with the situation where multiple objects, possibly in different categories (e.g., cars, pedestrians, etc.), may be present in one image. The goal of the detection is to find objects of interest in the image. As a result of object detection, each detected object is labeled with a bounding box. An example is shown in Fig.10.1.

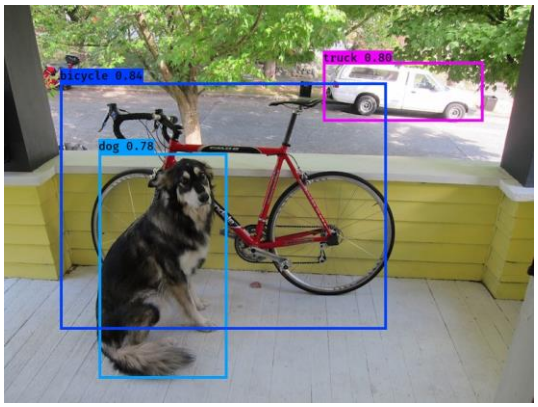


Fig.10.1 An example of object detection (generated by YOLO v2)

A bounding box is a rectangle that surrounds an object, that specifies its location, class (e.g., dog, bicycle) and confidence (how likely the object is present in the box). For example, the bounding box for the dog in Fig.10.1 shows the location of the dog and a confidence of 0.78. We can numerically specify the rectangle either by 1) the coordinates of its top-left corner and bottom right corner; or 2) the coordinate of its center and its width and height. In our context, we adopt the second one: [x,y, h, w], where the center of the bounding box is at (x, y), and the height and width of bounding box are h and w, respectively. The coordinate of the up-left corner of the image is (0,0) while the coordinate of the bottom right corner is (1,1).

10.2 YOLO (v1)

YOLO (You Only Look Once) (v1) was proposed by (Redmon, J. *et al*., 2015), which achieved object detection with a speed of 150 FPS (frames per second) in video streaming and a good mean average precision (mAP) (63%) on PASCAL VOC 2007 dataset (see the definition of mAP in Section 10.6). Later, YOLO (v2) and YOLO 9000 were proposed by (Redmon, J. *et al*., 2016), which at 67 FPS gave mAP of 76.8% on VOC 2007 dataset. Two years later, Furthermore, in 2018, YOLO (v3) (Redmon, J. *et al*., 2018) was released with further improved object detection accuracy and speed. YOLO (v3) introduced a new backbone architecture, called Darknet-53, which improved feature extraction and added additional anchor boxes to better detect objects at different scales. After YOLO (v3), a few groups developed different versions of YOLO for further improvements, up to YOLO (v8) in 2023. In this chapter, we will explore the details of YOLO v1 to v3.

The generic architecture of different YOLO versions can be illustrated in Fig.10.2. In general, a YOLO architecture consists of three parts: backbone, neck, and head. The backbone is a deep convolutional neural network that acts as a feature extractor. The backbone is usually pre-trained on a classification image dataset (e.g., ImageNet). The neck is a feature collector that collects feature maps from different stages of the backbone. The purpose of the neck is to enhance the semantic representation and richness of features extracted for objects of various shapes and sizes. The head is used for loss calculation during training and prediction during inference. Non-max suppression is an algorithm to select the best bounding box for an object and reject redundant bounding boxes.

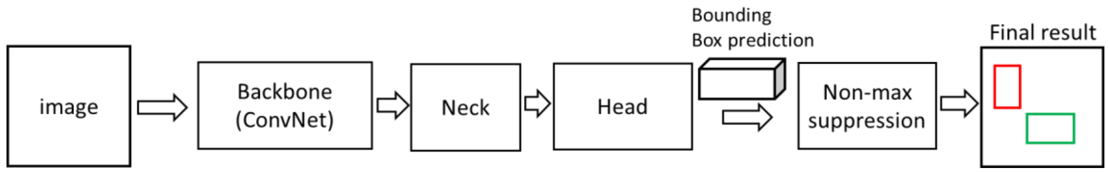


Fig.10.2 the generic architecture of YOLO

Table 10.1 YOLO (v1) architecture (input image size: $448 \times 448 \times 3$)

Layers	filters	Size/stride	output	notes
Conv	64	$7 \times 7/2$	$224 \times 224 \times 64$	
Maxpool		$2 \times 2/2$	$112 \times 112 \times 64$	
Conv	192	$3 \times 3/1$	$112 \times 112 \times 192$	
Maxpool		$2 \times 2/2$	$56 \times 56 \times 192$	
Conv	128	$1 \times 1/1$	$56 \times 56 \times 128$	
Conv	256	$3 \times 3/1$	$56 \times 56 \times 256$	
Conv	256	$1 \times 1/1$	$56 \times 56 \times 256$	
Conv	512	$3 \times 3/1$	$56 \times 56 \times 512$	
Maxpool		$2 \times 2/2$	$28 \times 28 \times 512$	backbone
Conv	256	$1 \times 1/1$	$28 \times 28 \times 256$	
Conv	512	$3 \times 3/1$	$28 \times 28 \times 512$	
Conv	256	$1 \times 1/1$	$28 \times 28 \times 256$	
Conv	512	$3 \times 3/1$	$28 \times 28 \times 512$	
Conv	256	$1 \times 1/1$	$28 \times 28 \times 256$	
Conv	512	$3 \times 3/1$	$28 \times 28 \times 512$	
Conv	256	$1 \times 1/1$	$28 \times 28 \times 256$	
Conv	512	$3 \times 3/1$	$28 \times 28 \times 512$	
Conv	512	$1 \times 1/1$	$28 \times 28 \times 512$	
Conv	1024	$3 \times 3/1$	$28 \times 28 \times 1024$	
Maxpool		$2 \times 2/2$	$14 \times 14 \times 1024$	
Conv	512	$1 \times 1/1$	$14 \times 14 \times 512$	
Conv	1024	$3 \times 3/1$	$14 \times 14 \times 1024$	
Conv	512	$1 \times 1/1$	$14 \times 14 \times 512$	
Conv	1024	$3 \times 3/1$	$14 \times 14 \times 1024$	
Conv	1024	$3 \times 3/1$	$14 \times 14 \times 1024$	
Conv	1024	$3 \times 3/2$	$7 \times 7 \times 1024$	
Conv	1024	$3 \times 3/1$	$7 \times 7 \times 1024$	
Conv	1024	$3 \times 3/1$	$7 \times 7 \times 1024$	
FC			4096	
FC			1470	neck
reshape			$7 \times 7 \times 30$	head

10.2.1 Architecture of YOLO v1

YOLO v1 has 24 convolutional layers followed by 2 fully connected layers (FC), as specified in Table 10.1. Leaky ReLU activation is used for all layers except the final layer that uses a linear activation function. The last convolutional layer generates a tensor with a shape (7, 7, 1024), shown in Fig.10.3. This tensor is then flattened to feed the subsequent fully connected layer. The last fully connected layer generates the final prediction tensor with a shape (7, 7, 30), i.e., 2 bounding box predictions per location, given 20 classes in the dataset, shown in Fig.10.4.

The backbone is composed of 24 convolutional layers and 4 Maxpooling layers. It generates a feature map tensor with a shape of $(7,7,1024)$, from an input image of $(448,448,3)$. Each cell in the feature map tensor, with a shape of $(1,1,1024)$, is the extracted feature vector corresponding to one grid cell in the image, illustrated in Fig.10.3.

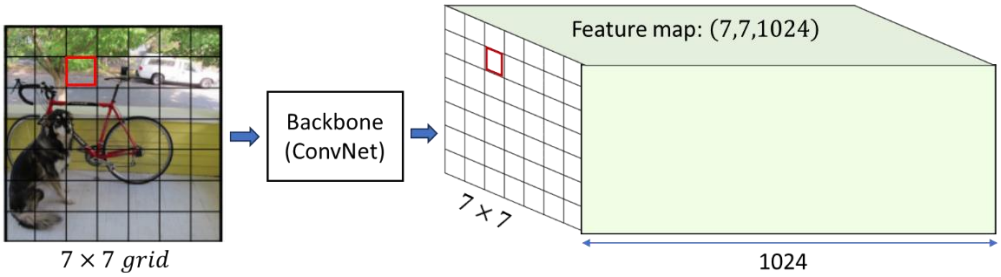
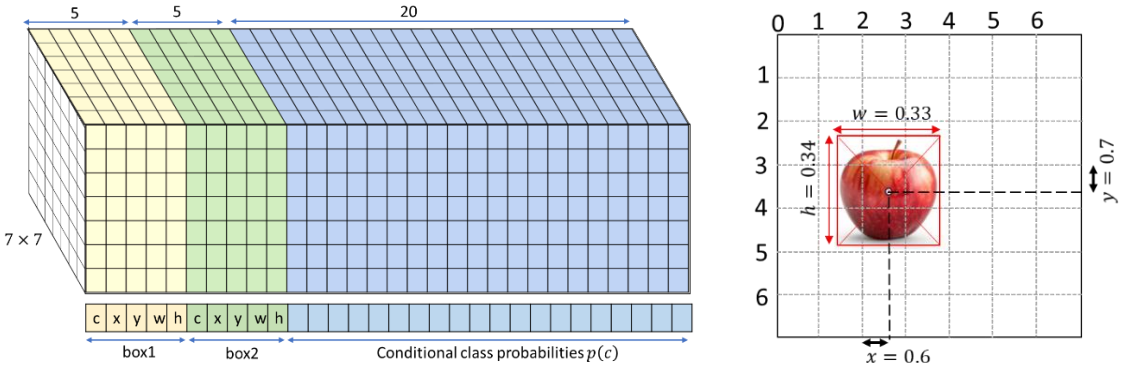


Fig.10.3 the backbone extracts one feature vector for each cell in the image grid (7 × 7).



(a) tensor shape (b) example for x,y,w,h

Fig.10.4 the prediction tensor of YOLO v1.

The feature map, output by the backbone convolutional neural network, is then passed through two fully connected layers, which delivers the bounding box prediction on each grid cell. The prediction is a tensor of $(7,7,30)$, illustrated in Fig.10.4 (a). The prediction tensor can be viewed as 49 vectors, and each vector has 30 elements. Each vector is responsible for predicting two bounding boxes for one grid cell of the image. Each bounding box is specified by five parameters: confidence (c) (or objectness score), box center (x,y) and its size (w,h). The confidence score reflects how confident the model is that the box contains an object and also how accurate it thinks the box is that it predicts. If no object exists in the cell, the confidence should be zero. If an object is predicted,

the confidence represents the IOU between the predicted box and any ground truth box. The coordinates (x, y) represent the center of the box relative to the bounds of the grid cell. The width (w) and height (h) are normalized with respect to the whole image. Thus, $x, y, w, h \in [0,1]$. Fig.10.4 (b) illustrates an example in which an apple is detected in the grid cell (2,3) (i.e., column 2 and row 3).

Each cell also predicts a set of class probabilities, conditioned on the grid cell containing an object, $p(c|object)$, c=1,2,…20, where 20 is the total number of classes.

At test time, the class-specific confidence scores for each box are the products of the conditional class probability and the box confidence. Note that YOLO v1 predicts only one object per grid cell.

For a convenience of further discussions on different YOLO versions, we denote grid size as $S \times S$, B as the number of bounding boxes per grid cell, and C as the number of classes. Thus, the shape of prediction tensor for YOLO v1 is $S \times S \times (5 \times B + C) = 7 \times 7 \times (5 \times 2 + 20)$.

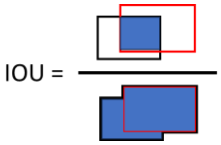
10.2.2 Training and loss function

Before training the entire architecture of YOLO v1, the authors pretrained the first 20 convolutional layers followed by an average-pooling layer and a fully connected layer, for classification on ImageNet 1000 classes, with input size of 224×224, achieving a top-5 accuracy of 88%.

Then, they convert the pretrained architecture for object detection by discarding the average pool and the fully connected layer and adding four convolutional layers and two fully connected layers with randomly initialized weights. Thus, the resulting architecture is specified in Table 10.1, with the first 20 convolutional layers pretrained.

The model is further trained for object detection on datasets from PASCAL VOC 2007 and 2012, based on the loss function (discussed below), with the input resolution increased to 448×448. It is essential to understand the loss function used for YOLO training. Object detection can be treated as a regression problem of target area prediction and category prediction. Specifically, the loss function of YOLO v1 can be divided into three parts: localization loss, confidence loss, and classification loss.

YOLO v1 predicts two bounding boxes per grid cell. For training purposes, we only want one bounding box predictor to be responsible for each object. We assign one box to be "***responsible***" for predicting an object based on which box has the highest current IOU (i.e., Intersection Over Union) with the ground truth. The IOU between two boxes is defined as the ratio of their intersection area to their union area. IOU with a value "1" implies a perfect match while a small value close to zero means a small overlap. Thus, IOU can be used to measure how close the predicted bounding box is to the ground truth bounding box, and also can be applied to detect the redundant bounding boxes for one object in non-max suppression (discussed later).



(10.1)

A) Localization loss

The localization loss measures the errors in the predicted boundary box locations and sizes. It only penalizes the bounding box localization error if that box is "responsible" for the ground truth box (i.e., the box has the highest IOU with the ground truth box in that grid cell).

$$\lambda_{coord}\sum_{i=1}^{S^2}\sum_{j=1}^{B}\mathbb{I}_{ij}^{obj}\left[\left(x_i-\hat{x}_i\right)^2+\left(y_i-\hat{y}_i\right)^2\right]$$

$$+\lambda_{coord}\sum_{i=1}^{S^2}\sum_{j=1}^{B}\mathbb{I}_{ij}^{obj}\left[\left(\sqrt{w_i}-\sqrt{\hat{w}_i}\right)^2+\left(\sqrt{h_i}-\sqrt{\hat{h}_i}\right)^2\right] \qquad (10.2)$$

where

$\mathbb{I}_{ij}^{obj}=1$ if the j-th bounding box in cell i is responsible for detecting the object, otherwise 0. In other words, localization error is added to loss only for the responsible boxes.

The hat over a variable indicates that the variable is a prediction (e.g., $\hat{x}_i$). A variable without a hat is a ground truth (e.g., x_i).

λ_{coord} (Default is set to 5) increases the weight for the loss in bounding box coordinates.

Sum-squared error equally weights errors in large boxes and small boxes. The desired loss should reflect that the same deviation in large boxes matters less than in small boxes. To partially address this, we predict the square root of the bounding box width and height instead of the width and height directly.

B) Confidence loss

For the boxes that are responsible for detecting an object, the confidence loss is

$$\sum_{i=1}^{S^2}\sum_{j=1}^{B}\mathbb{I}_{ij}^{obj}\left(C_i-\hat{C}_i\right)^2 \qquad (10.3a)$$

where

$\hat{C}_i$ is the box confidence score for the box j in cell i. Note that the ground truth box confidence score C_i for the box with an object is defined to be IOU_{pred}^{truth} by the original paper. However, the ground truth confidence score is simply set to 1 in many implementation examples.

For the boxes that are not responsible for detecting an object, the confidence loss is

$$\lambda_{noobj}\sum_{i=1}^{S^2}\sum_{j=1}^{B}\mathbb{I}_{ij}^{noobj}\left(C_i-\hat{C}_i\right)^2 \qquad (10.3b)$$

where

$\mathbb{I}_{ij}^{noobj}$ is the complement of $\mathbb{I}_{ij}^{obj}$, i.e., that the box j in grid cell i is not responsible for detection.

$\hat{C}_i$ is the box confidence score for the box j in cell i. Note that the ground truth box confidence score C_i for the box without an object is defined to be 0.

λ_{noobj} (=0.5 by default) decreases the loss from confidence predictions for boxes that don't contain objects. Since most boxes do not contain any objects, we train the model to detect background more frequently than detecting objects. To remedy this imbalance, λ_{noobj} is used to limit the loss from background detections.

C) Classification loss

If an object appears in cell i, the sum-squared error of the predicted class conditional probabilities in cell i is counted for the loss function,

$$\sum_{i=1}^{S^2} \mathbb{I}_i^{obj} \sum_{c \in classes} \left(p_i(c) - \hat{p}_i(c) \right)^2 \tag{10.4}$$

where

$\mathbb{I}_i^{obj}$=1 if an object appears in cell i. This implies that $p_i(c)$ is conditional.

$\hat{p}_i(c)$ denotes the predicted conditional class probability for class c in cell i. Given that the ground truth object belongs to a class, the ground truth probability is 1.0 for that class, and 0.0 for all others.

D) Total loss

The total loss function is given by the sum of (10.2), (10.3a), (10.3b) and (10.4), i.e.,

$$\begin{aligned} Loss = {} & \lambda_{coord} \sum_{i=1}^{S^2} \sum_{j=1}^{B} \mathbb{I}_{ij}^{obj} \left[\left(x_i - \hat{x}_i \right)^2 + \left(y_i - \hat{y}_i \right)^2 \right] \\ & + \lambda_{coord} \sum_{i=1}^{S^2} \sum_{j=1}^{B} \mathbb{I}_{ij}^{obj} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right] \\ & + \sum_{i=1}^{S^2} \sum_{j=1}^{B} \mathbb{I}_{ij}^{obj} \left(C_i - \hat{C}_i \right)^2 + \lambda_{noobj} \sum_{i=1}^{S^2} \sum_{j=1}^{B} \mathbb{I}_{ij}^{noobj} \left(C_i - \hat{C}_i \right)^2 \\ & + \sum_{i=1}^{S^2} \mathbb{I}_i^{obj} \sum_{c \in classes} \left(p_i(c) - \hat{p}_i(c) \right)^2 \end{aligned} \tag{10.5}$$

Note that the loss function penalizes classification error only if an object is present in that grid cell. It also only penalizes bounding box coordinate error if that box is responsible for the ground truth box. It penalizes the confidence score errors for both object-present boxes and non-object-present boxes, but with different weights (non-object-present boxes have less weight).

The model, with the first 20 convolutional layer pretrained, was further trained for object detection on datasets PASCAL VOC 2007 and 2012, based on the loss function (10.5). The detailed training settings and tricks can be found in the original paper (Redmon, J. *et al*., 2015).

10.2.3 Inference and non-maximal suppression (NMS)

In inference, the YOLO v1 network predicts confidence score, center coordinates, width and height for each bounding box and conditional class probabilities for each grid cell, as shown in Fig.10.4. There are totally 98 bounding boxes. Finally, non-max suppression (NMS) is performed for each class separately to discard the redundant detected boxes while keeping the best one for a detected object.

Consider an example in Fig.10.5. We assume that only one class (e.g., car) is considered in our discussion for simplicity. In this example five bounding boxes have confidence scores above a pre-defined threshold (e.g., 0.6). Obviously, three of them are associated with one car while the other two bounding boxes with another car, as shown in Fig.10.5 (left). After non-max suppression, only two best bounding boxes, one per car, should be kept, as shown in Fig.10.5 (right). The NMS algorithm rejects those bounding boxes which are too close to a high confident bounding box.

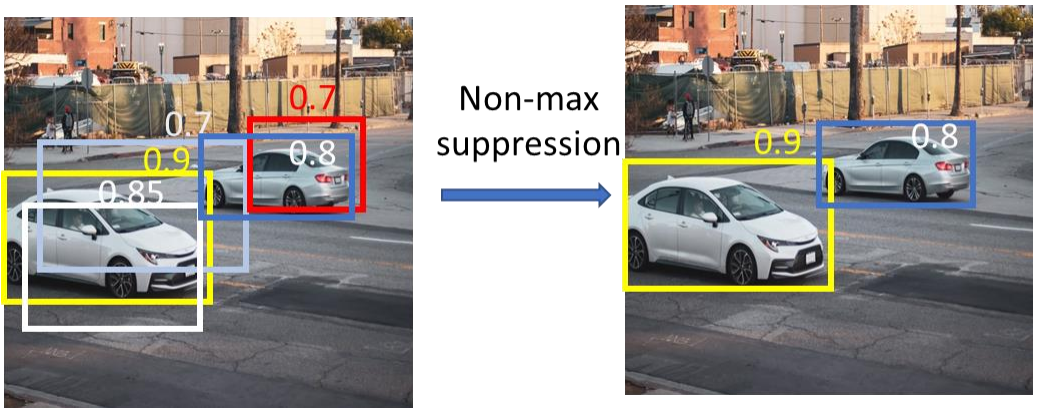


Fig.10.5 Non-max suppression (the number associated with each box is confidence score).

The non-max suppression algorithm can be described below.

Algorithm: non-max suppression

Given: the output tensor (i.e., a list of bounding boxes), a pre-defined threshold for confidence score, a pre-defined threshold for IOU.

Output: a final list that consists of qualified bounding boxes.

Step 1: Remove any bounding box with confidence less than a threshold (e.g., 0.6). Put the remaining boxes into a list, called *remain_list*, in descending order of confidence score.

Step 2: Do the following by looping over *remain_list* until *remain_list* is empty:

1. From *remain_list*, move the first box (i.e., one with the highest confidence), b_{max}, to *final_list*.
2. From *remain_list*, delete any remaining box (for the same class as b_{max}) with IOU≥ a threshold (e.g. 0.5) with the output box, b_{max}.

10.3 YOLO (v2)

YOLO v2 (or YOLO9000) is the second version of the YOLO with the objective of improving the accuracy significantly while making it faster. Compared to YOLO v1, YOLO v2 makes the improvements through the following aspects:

- Batch normalization. Batch normalization leads to significant improvements in convergence while eliminating the need for other forms of regularization (e.g., dropout). Batch normalization is added to all convolutional layers in YOLO v2, leading to 2% mAP improvement.
- Higher resolution classifier. YOLO v1 was pre-trained for classification network by images with resolution 224×224, and then for detection by images with resolution 448×448. In YOLO v2, the classifier is fine-tuned by the resolution 448×448, which leads to 4% mAP improvement.
- Architecture for fine-grained features and more bounding boxes per grid cell.
- Anchor boxes. YOLO v2 doesn't predict the bounding box parameters directly, instead it predicts the pre-defined anchor boxes.
- Multi-scale training.

YOLO v2 gives state-of-the-art detection accuracy on PASCAL VOC and COCO. It can run on varying sizes offering a tradeoff between speed and accuracy. At 67 FPS, YOLO v2 can give an accuracy of 76.8 mAP while at 40 FPS the detector gives an accuracy of 78.6 mAP, on VOC dataset.

10.3.3 Architecture of YOLO v2

The overall architecture of YOLO v2 is described in Table 10.2. YOLO v2 adopts a model, called Darknet-19, for its backbone network.

Darknet-19 has 19 convolutional layers and 5 maxpooling layers. Inspired by GoogleNet and Network-in-Network, we use 3×3 filters and 1×1 filters alternatively. The 1×1 filters compress the feature representation between 3×3 convolutional layers. The number of channels is doubled while the feature map size is halved, by each maxpooling layer. This results in an output feature map size 13×13 for an input image 416×416 after 5 maxpooling layers (note: $\frac{416}{2^5} = 13$).

In Darkent-19, the last convolutional layer $1 \times 1 \times 1000$ and the avgpool layer are only used to train the backbone on ImageNet 1000 dataset. Please note that it was initially trained on images 224×224, and then fine-tuned on images 448×448.

To construct YOLO v2 network for object detection, we modify the pretrained Darknet-19 by removing the last convolutional layer and the avgpool layer, and adding 3 convolutional layers with filter 3×3 and 1024 channels per layer and a final 1×1 convolutional layer with 125 channels, as described in Table 10.2. The 125 channels are responsible for 5 bounding boxes. In addition, the output from the final $3 \times 3 \times 512$ layer is reshaped to $13 \times 13 \times 2048$, and concatenated with the output of the third to the last convolutional layer so that the model can use fine grain features, as shown in Fig.10.6.

Table 10.2 YOLO v2 architecture (input image size: $416 \times 416 \times 3$)

Layers	filters	Size/stride	Output
Conv	32	3×3	416×416
Maxpool		$2 \times 2/2$	208×208
Conv	64	3×3	208×208
Maxpool		$2 \times 2/2$	104×104
Conv	128	3×3	104×104
Conv	64	1×1	104×104
Conv	128	3×3	104×104
Maxpool		$2 \times 2/2$	52×52
Conv	256	3×3	52×52
Conv	128	1×1	52×52
Conv	256	3×3	52×52
Maxpool		$2 \times 2/2$	26×26
Conv	512	3×3	26×26
Conv	256	1×1	26×26
Conv	512	3×3	26×26
Conv	256	1×1	26×26
Conv	512	3×3	26×26
Maxpool		$2 \times 2/2$	13×13
Conv	1024	3×3	13×13
Conv	512	1×1	13×13
Conv	1024	3×3	13×13
Conv	512	1×1	13×13
Conv	1024	3×3	13×13

Backbone

Reshape to $(13 \times 13 \times 2048)$

For backbone pre-training			
Layers	Filters	Size/stride	output
Conv	1000	1×1	13×13
Avgpool		Global	1000
Softmax			

For detection			
layers	filters	Size/stride	output
Conv	1024	3×3	13×13
Conv	1024	3×3	13×13
Concat	3072		13×13
Conv	1024	3×3	13×13
Conv	125	1×1	13×13

The YOLO v2 model was trained on COCO and VOC datasets. For an input image 416×416, YOLO v2 divides it into an 13×13 grid and predicts 5 bounding boxes per grid cell. Each bounding box prediction includes 25 elements: 1 for confidence score, 4 for coordinates, and 20 for conditional class probabilities of 20 classes. Thus, the prediction tensor of YOLO v2 has a shape of $13 \times 13 \times 125$, as illustrated in Fig.10.7. In general, the YOLO v2 prediction tensor has a shape of $13 \times 13 \times \big(B \times (5 + C)\big)$, where B is the number of bounding boxes per grid cell, C is the total number of classes.

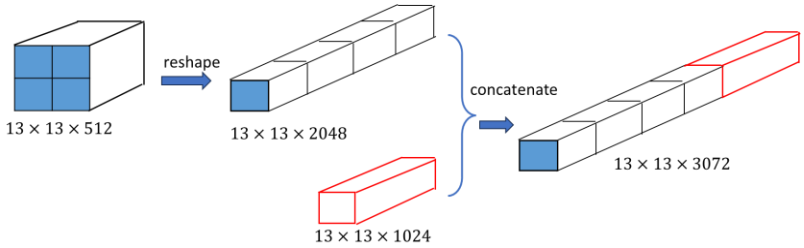


Fig.10.6 Concatenation for the passthrough layer

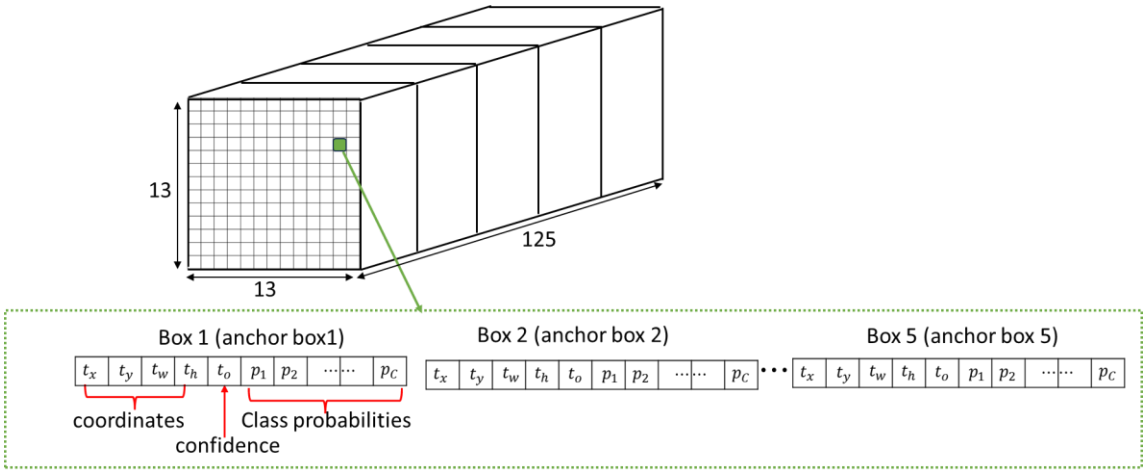


Fig.10.7 Prediction tensor of YOLO v2

10.3.2 Anchor boxes

An important ingredient of YOLO v2 is the concept of anchor boxes, which improves performance and has been applied in the subsequent YOLO models. YOLO v1 predicts the coordinates of bounding boxes directly using fully connected layers on top of the convolutional feature extractor. Instead of predicting coordinates directly, YOLO v2 predicts a bounding box relative to a pre-defined bounding box, called *anchor box*. In other words, the model uses the anchor boxes as the initial values of bounding boxes, and then learns to adjust the bounding box for fitting the detected object closely. This idea makes it easier for the model to learn. Furthermore, the use of a set of anchor boxes enables a model to detect multiple objects, objects of different scales, and overlapping objects.

The number of anchor boxes and their aspect ratios can be determined by a statistical analysis on the target datasets. For example, *k*-means clustering on the dimensions of objects in VOC and COCO datasets suggests that k = 5 (i.e., 5 different shapes for anchor boxes) give a good tradeoff between the recall and the complexity. As the result, a widely used set of anchor boxes (p_w, p_h) is given in yolov2.cfg (provided by YOLO v2 paper authors) as {(0.57273, 0.677385), (1.87446, 2.06253), (3.33843, 5.47434), (7.88282, 3.52778), (9.77052, 9.16828)}, drawn in Fig.10.8. Note that p_w, p_h are the width and the height, respectively, and they are normalized by the grid cell size. Alternatively, the size of bounding box or anchor box is usually normalized by the image size in YOLO algorithms. Thus, divided by 13, the above set of

anchor boxes can be specified, relative to the image size, as anchors={(0.04405615, 0.05210654), (0.14418923, 0.15865615), (0.25680231, 0.42110308), (0.60637077, 0.27136769), (0.75157846, 0.70525231)}.

The geometric relationship between an anchor box and the corresponding bounding box is defined by eq (10.6) and illustrated in Fig.10.9.

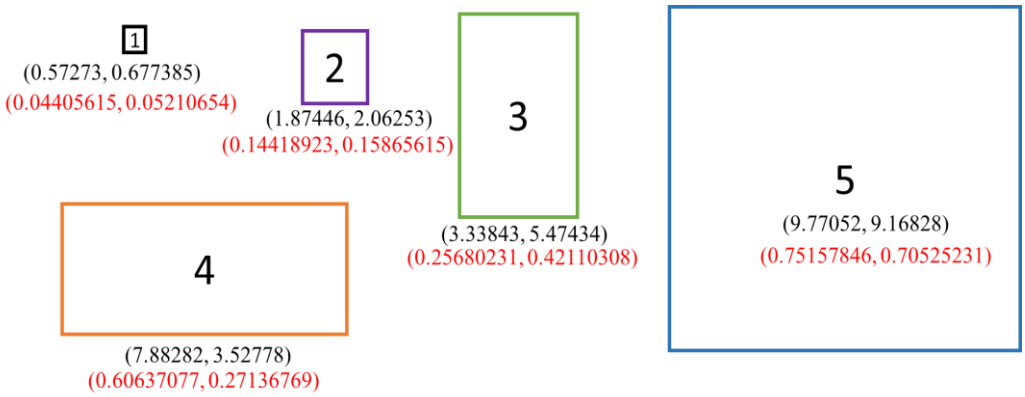


Fig.10.8 Five anchor boxes in YOLO v2

(normalized by the grid cell size in black [yolov2.cfg by YOLO v2 authors])

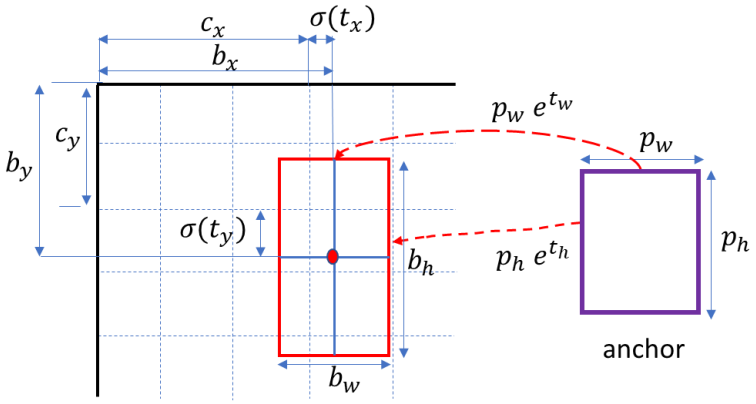


Fig.10.9 Bounding box prediction from the output tensor of YOLO v2

10.3.3 Predictions from YOLO v2

The prediction tensor $(13 \times 13 \times 125)$ generated by YOLO v2 model is illustrated in Fig.10.7. To understand the predictions, we imagine that the image is divided into a 13×13 grid. For each grid cell, there is a prediction 125-element vector consisting of 5 bounding boxes, which corresponds to 5 anchor boxes, respectively. The prediction of each bounding box includes box coordinates (4 numbers), confidence (1 number) and conditional class probabilities (20 numbers).

Now let's focus on the bounding box prediction, corresponding to an anchor box(p_w, p_h) in a specific grid cell (c_x, c_y), where $0 \le c_x \le 12, 0 \le c_y \le 12$ are the index of the grid cell, for

instance, cell(3,2) is the grid cell where the bounding box center point is located, as illustrated in Fig.10.9.

Suppose t_x, t_y, t_w, t_h, t_o are the data from the prediction tensor of YOLO v2. Then, the actual parameters of the bounding box can be obtained by the following transformation:

$$b_x = \sigma(t_x) + c_x \tag{10.6 a}$$

$$b_y = \sigma(t_y) + c_y \tag{10.6 b}$$

$$b_w = p_w e^{t_w} \tag{10.6 c}$$

$$b_h = p_h e^{t_h} \tag{10.6 d}$$

$$objectscore = P_r(object) \times IOU(b, object) = \sigma(t_o) \tag{10.6 e}$$

where

$\sigma(\square)$ is the sigmoid function.

b_x, b_y are the coordinates of the bounding box center point.

b_w, b_h are the width and the height of the bounding box.

Note that $\sigma(t_x), \sigma(t_y) \in (0,1)$ are the offsets of the bounding box center point relative to the left-top corner of the grid cell. t_w, t_h are used to scale the anchor box to get the bounding box size. Object score represents the probability that an object is contained inside a bounding box, and it is obtained by passing t_o through a sigmoid function. Note that the box position parameters such as $c_x, c_y, b_x, b_y, b_w, b_h, p_w, p_h$ are numerically normalized relative to the grid size (width or height), i.e., their units are the grid size.

The loss function defined in equation (10.5) for YOLO v1 can be applied to YOLO v2, in general. Since anchor boxes are used for bounding box prediction, the ground truth parameters used in the loss function can be obtained by the inverse transform of (10.6) based on the label bounding boxes.

10.4 YOLO (v3)

YOLO (v3) introduced a new backbone architecture, called Darknet-53, which improved feature extraction and added additional anchor boxes to better detect objects at different scales. This model features multi-scale detection, a stronger feature extraction network, and a few changes in the loss function.

10.4.1 Architecture of YOLO v3

The neural network of YOLO v3 consists of three parts: Darknet-53, neck, and detection head, as shown in Fig.10.10. Darknet-53 is used as the backbone to extract features from the input images. The combination of Darknet-53 and the up-sample neck network results in a feature pyramid network in which a feature map gradually decreases in spatial dimension but increases later again and is concatenated with previous feature maps with corresponding sizes. The different sized feature maps are then fed to a distinct detection head. Each head is implemented by a few convolutional layers and generates predictions at a different grid scale.

Our presentation is based on the following settings for YOLO v3: three scale predictions (13x13, 26x26, 52x52 for an input image 416x416, three anchor boxes (B=3) for each scale prediction, and the number of classes C. The prediction for each grid cell is a vector with M elements, where $M = B \times (5 + C)$. Thus, M=75 for C=20 while M=255 for C=80.

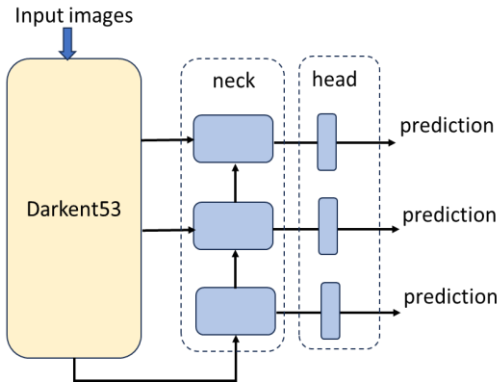
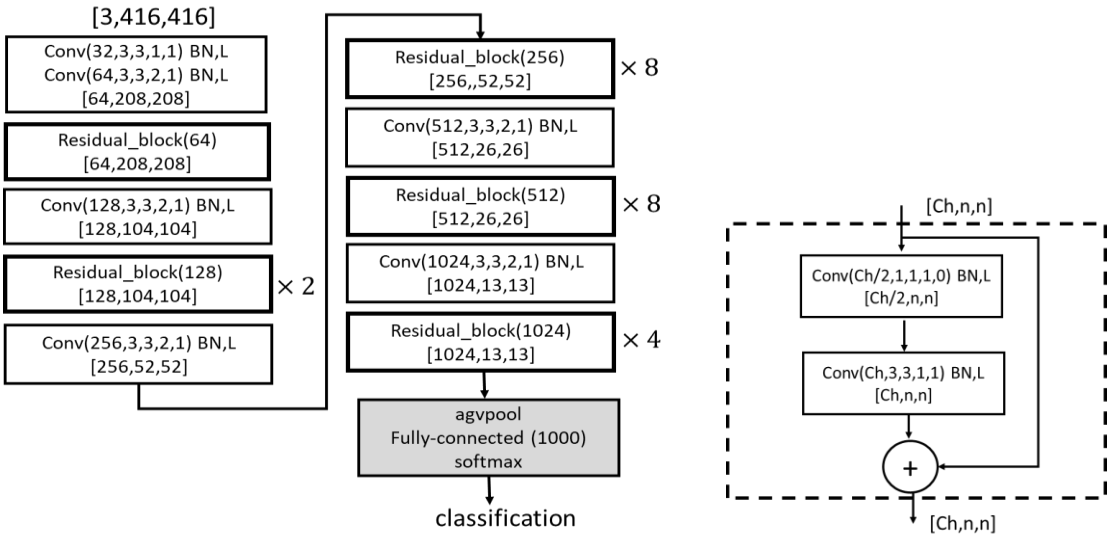


Fig.10.10 Overall architecture of YOLO v3

A) Backbone: Darknet-53

The backbone network, Darknet-53, is illustrated in Fig.10.11. Darknet-53 has 53 trainable layers (52 convolutional layers and one fully connected layer). The last three layers (avgpool, full connected, softmax) are designed for pre-training, and thus being removed when it is adopted for YOLO feature extraction. Darknet-53 was pre-trained on ImageNet dataset with the input size 256×256. In Fig.10.11, the shape of the tensor from each block is specified as [channel, w, h] while a convolutional layer is specified as **Conv**(*channel*, *kernel size*, *kernel size*, *stride*, *padding*). Note that all convolutional layers are followed by batch normalization (BN) and leakyReLU activation (L).



(a) architecture of Darknet-53 (b) residual_block(Ch)

Fig.10.11 Darknet-53

Darknet-53 has 23 residual blocks. Each residual block contains one 1 ×1 and one 3×3 convolutional layer. At the end of each residual unit, an element-wise addition is performed between the input tensor and the output tensor from the second convolutional layer. The shape of the output tensor of a residual block is the same as the input tensor.

The down-sample step is performed by five separate convolutional layers with a stride of 2.

B) Neck: up-sample network

To detect objects at different scales, YOLO v3 adopts a Feature Pyramid Network (FPN) to extract the feature maps at different grid scales. The architecture of YOLO v3 is detailed by Fig.10.12 where we list all convolutional layers (or blocks) and the corresponding output tensor shapes, assuming that the input is a batch of images, i.e., [N, 3, 416, 416]. All convolutional layers, except the last Conv2d layer in each detection head, are followed by a batch normalization and leakyReLU. We count any one of the following items as one layer: convolutional layer, shortcut pass, input route from a distant layer, upsample (), concatenate (after upsample), and yolo detection. Thus, a residual block counts for 3 layers. The entire YOLO model consists of 107 layers. All layers are labeled in an order from layer 0 to layer 106.

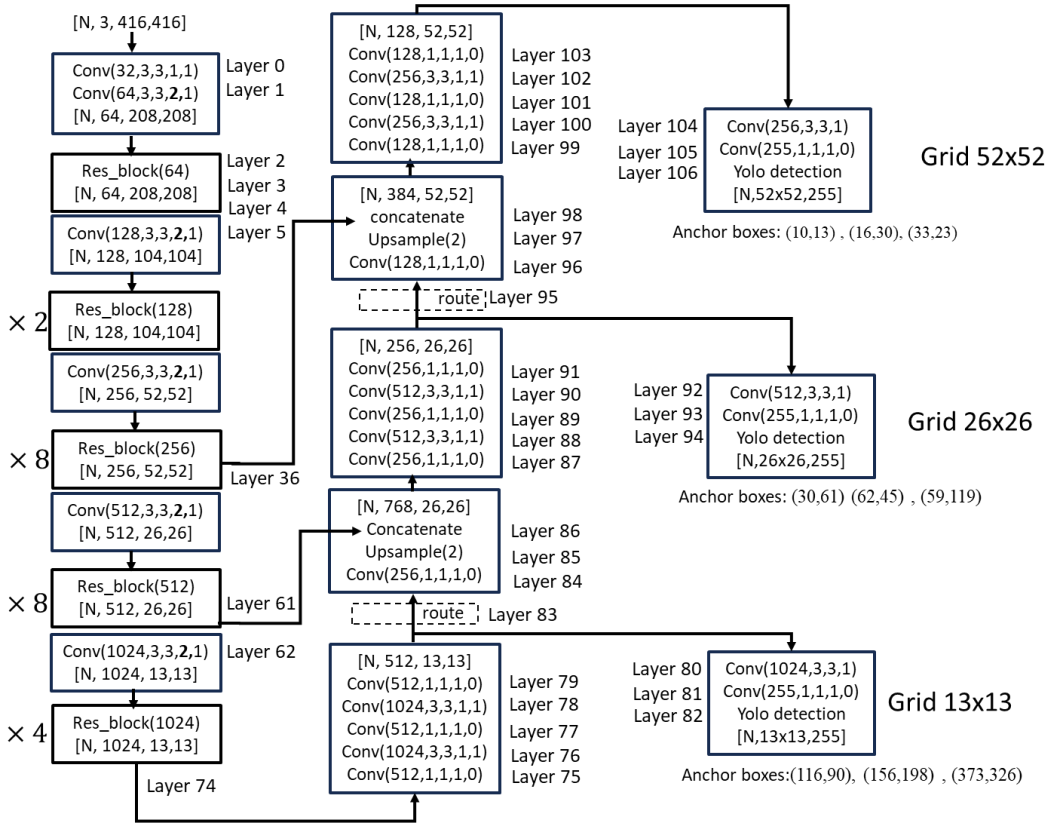


Fig.10.12 Architecture of YOLO v3 (all conv layers are followed by BN and leakyReLU, except that the last conv layer in each detection head. We define the first conv layer as layer 0 and count three layers for each Res_block).

The entire Darknet-53 down-samples the input image 416×416 by a factor of 32, and thus generates feature maps at the grid scale 13×13. A set of convolutional layers (layers 75 to layers 82: 1×1 and 3×3 alternatively) process the feature map and generate the prediction [N,13,13, M] at grid scale 13×13. Note that M=255 when C=80, for instance.

Layers 83 to 94 are responsible to generate the prediction at grid scale 26×26. First, we concatenate the feature map [N,512,26,26] from the last residual block (512) in Darknet-53 and the up-sampled (with a factor of 2) feature map [N,256,26,26] from the neck of scale 13×13. Then the merged feature map [N,768,26,26] will be passed through a series of convolutional layers, which are similar to those convolutional layers used for the scale 13×13, but the number of channels is halved. The prediction is a tensor [N,26,26, M]. The up-sample operation is illustrated in Fig.10.13.

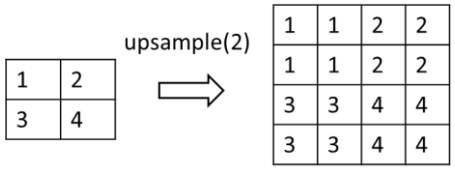


Fig.10.13 Up-sample by a factor of 2 using "nearest mode (default)"

Similarly, layers 95 to 106 are designed to predict at grid scale 52×52. Specifically, we concatenate the feature map [N,256,52,52] from the last residual block (256) in Darknet-53 and the up-sampled (with a factor of 2) feature map [N,128,52,52] from the neck of scale 26×26. Then the merged feature map [N,384,52,52] will be passed through a set of convolutional layers to generate the prediction [N,52,52, M].

C) Understand predictions at three grid scales.

YOLO v3 generates a prediction tensor at each grid scale. The tensor shapes are [N, 13, 13, M], [N, 26, 26, M], and [N, 52, 52, M] for three grid scales, respectively, where N is the batch size, $M = B \times (5 + C)$, B is the number of anchor boxes for one grid cell, C is the number of classes. YOLO v3 predicts three bounding boxes at each grid cell across all different scales (i.e., B=3). As an example, Fig.10.14 shows the prediction tensor at the scale 13×13 with C=80.

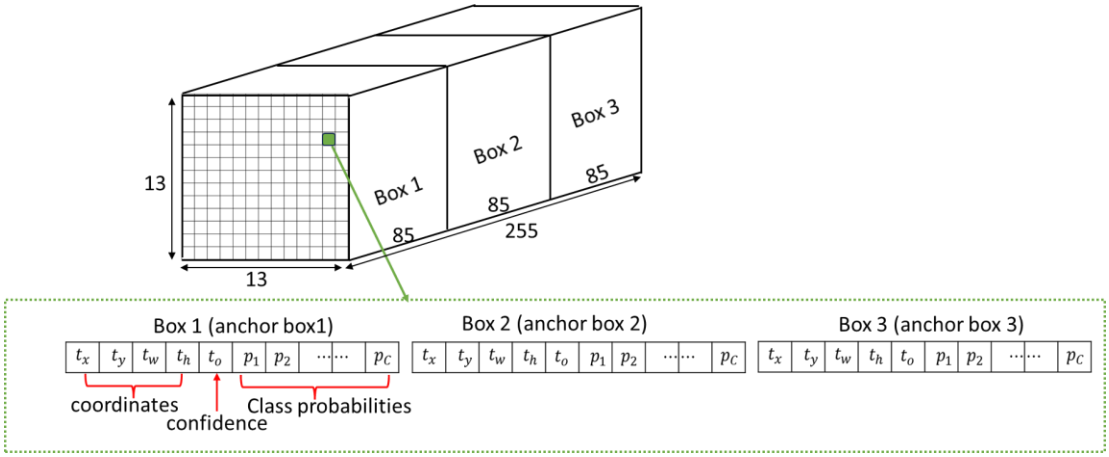


Fig.10.14 Prediction tensor at scale 13×13 for C=80.

To determine the anchor boxes, YOLO v3 applies *k*-means clustering on the COCO dataset and select 9 clusters. As a result, the width and height of the nine anchor boxes are: (10×13), (16×30), (33×23), (30×61), (62×45), (59×119), (116 × 90), (156 × 198), (373 × 326). These 9 anchor boxes are grouped into 3 different groups according to their size. The three smallest anchor boxes are assigned to the finer grid scale (e.g., 52x52), The three middle-sized ancho boxes are used for the middle fine grid scale (e.g., 26x26), and the three largest anchor boxes are used for the coarse grid scale (e.g., 13x13). Note that the anchor boxes can be normalized to the input image by dividing 416 or normalized to the grid size by dividing the corresponding stride (e.g., 32 for 13x13 grid scale).

With the information of anchor boxes, the final bounding box parameters and confidence score can be obtained by the transformation defined in equation (10.6). The relevant data from the last conv layers (i.e., layer 81, 93, 105 in Fig.10.12) need to pass a sigmoid function to form the class probabilities.

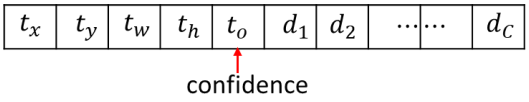
Like YOLO v2, the final step in the inference is to apply a non-max suppression on the prediction tensor to eliminate unqualified bounding boxes.

10.4.2 Loss function of YOLO v3

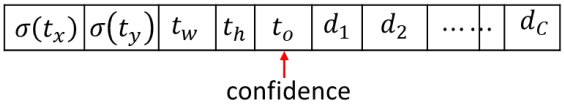
It is essential to understand the loss function and its implementation for training a YOLO model. Pre-trained YOLO v3 models are available in open resources for detecting pre-defined general 80 classes from COCO dataset. However, in many applications, it is required to train the model based on a customized dataset. In practice, we compute the loss at each grid scale independently, and then obtain the total loss by adding the losses at three grid scales.

A) Prediction

Suppose the prediction at a grid scale s is represented as a tensor with a shape (3, s, s, 85), where s is the grid size (e.g., 13, 26, 52), denoted as $pred(3, s, s, 85)$. Thus, one predicted bounding box, corresponding to anchor box *k* (k=0,1,2), at grid cell (i, j), is $pred[k, i, j, 0{:}\,85]$ that has a format:



Note that all numbers above are generated by the last Conv2d layer, without passing through any activation function or a sigmoid function. To match the format of the target (discussed later), it is convenient to update the prediction tensor, $pred(3, s, s, 85)$ to the following format, by applying a sigmoid function to t_x, t_y, tentatively,



The meanings of the first four items were illustrated in Fig.10.9.

B) Target

To compute the loss of a prediction for an input image, we need to generate the target (i.e., ground truth) based on the label bounding boxes associated with the image. The label bounding

boxes are given in a format per object: $[class, bx, by, bw, bh]$, where bx and by are the center of the box, bw, bh are the width and height of the box, and all of them are relative to the image, $class$ is the index of the object class. The target can be organized as a tensor with a shape (3, s, s, 6), illustrated in Fig.10.15 for s=13.

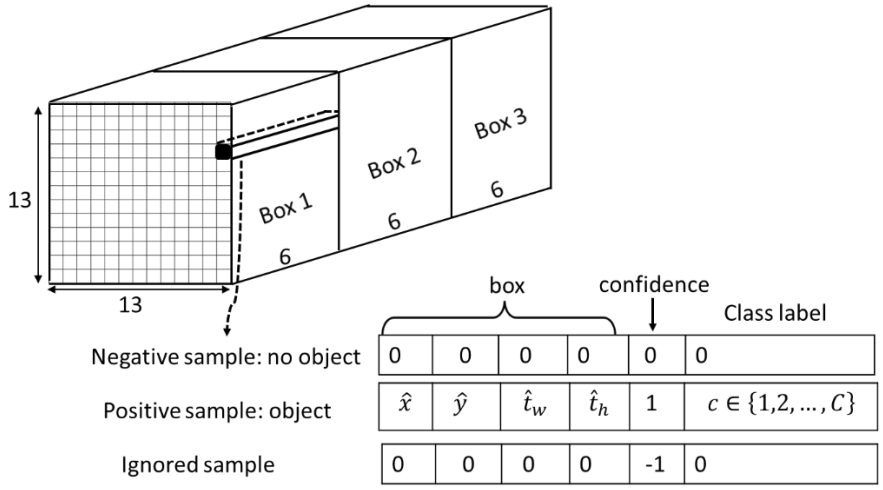


Fig.10.15 Target at grid scale 13

At each grid scale, we need to pick an anchor box responsible for the ground truth bounding box. The selected anchor box (positive sample) should be in the grid cell where the ground truth box is located, and also has the highest IOU with the ground truth box among the three anchor boxes in that cell. The confidence of the picked anchor box is assigned to 1. If any of the remaining two anchor boxes has a high (but not the highest) IOU with the ground truth box (e.g, more than an ignore_threshold), then they will be labeled as ignored box by assigning the confidence as -1, so that the corresponding prediction will be ignored, i.e., will not contribute any loss. All other target boxes (negative samples) are labeled as "no object" by assigning the confidence as 0.

In Fig.10.15, the elements in the target for positive samples are calculated as follows:

1) Ground truth box location offset to the up-left corner of the cell, relative to the grid cell size,

$$\begin{cases} \hat{x} = bx \cdot s - int(s \cdot bx) \\ \hat{y} = by \cdot s - int(s \cdot by) \end{cases} \tag{10.7}$$

2) Scaling parameters for width and height by the inverse function of eq. (10.6 c and 10.6 d)

$$\begin{cases} \hat{t}_w = \log\left(\frac{bw}{p_w}\right) \\ \hat{t}_h = \log\left(\frac{bh}{p_h}\right) \end{cases} \tag{10.8}$$

where p_w, p_h are the anchor box width and height relative to the image size, respectively.

Therefore, we generate the target tensor, denoted as *Target* (*3,s,s,6*), shown in Fig.10.15 at each grid scale. There are three types of target vectors (each vector has 6 elements): negative sample (no object), positive sample (object), ignored sample. In the target tensor, each 6-element vector corresponds to a predicted 85-element vector at a grid cell and a particular anchor box.

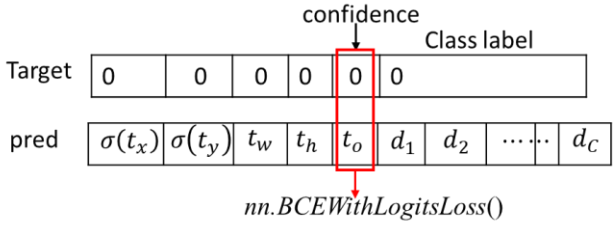
The loss function is defined to measure the mismatch between the target vectors and predicted vectors.

C) Loss function

Although Equation (10.5) was proposed for the loss function in YOLO v1, it defines the framework of loss function for the subsequent YOLO versions. There are many variants for the loss function and its implementation for YOLO v3. We introduce one of them below. The loss is composed of four parts: "no object" confidence loss, "object" confidence loss, "object" box loss, and "object" classification loss.

1) "no object" confidence loss. If the target has a confidence 0 (i.e., no object at the cell for the anchor box), the corresponding prediction only results in a confidence loss, specified as the binary cross-entropy loss,

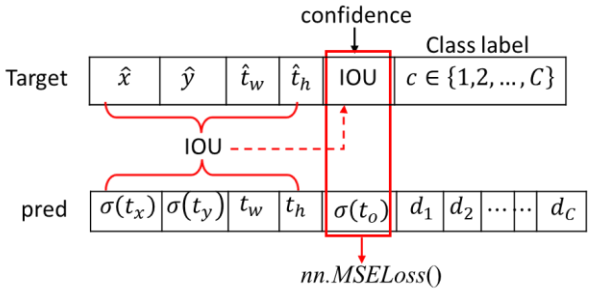
$$loss_{noobj} = -\lambda_{noobj} \sum_{i=0}^{s\times s} \sum_{j=0}^{3} 1_{ij}^{noobj} \log\left(1 - \sigma(t_o)\right) \tag{10.9}$$



In PyTorch, we can call *nn.BCEWithLogitsLoss*() to compute $loss_{noobj}$ using the confidence in the target (actually 0) and the prediction t_o, for all negative samples.

2) "object" confidence loss. For a positive sample, the confidence "1" in the target should be updated by the IOU between the target box and the predicted box, and then we compute the confidence loss as the mean squared error between the IOU and $\sigma(t_o)$,

$$loss_{obj} = \lambda_{obj} \sum_{i=0}^{s\times s} \sum_{j=0}^{3} 1_{ij}^{obj} \left(IOU - \sigma(t_o)\right)^2 \tag{10.10}$$

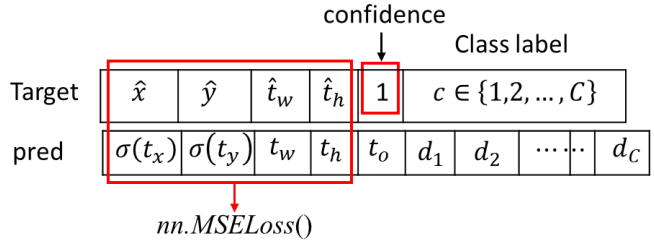


In PyTorch, we can call *nn.MSELoss*() to compute $loss_{obj}$, using the updated confidence scores (shown above) for all positive samples.

3) "object" box loss. For a positive sample, the box loss defines the location mismatch between the ground truth bounding boxes and the predicted bounding boxes. The box loss is specified as MSE loss,

$$loss_{box} = \lambda_{coord} \sum_{i=0}^{s\times s} \sum_{j=0}^{3} 1_{ij}^{obj} \left[\left(\hat{x} - \sigma(t_x)\right)^2 + \left(\hat{y} - \sigma(t_y)\right)^2 + (\hat{t}_w - t_w)^2 + (\hat{t}_h - t_h)^2\right]$$

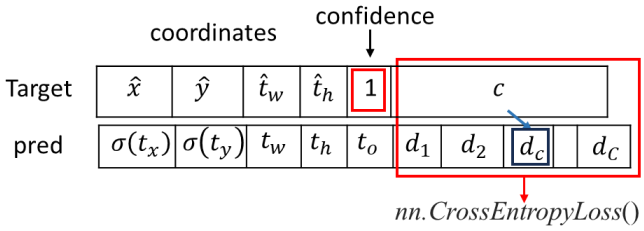
(10.11)



In PyTorch, we call *nn.MSELoss*() to compute $loss_{box}$, based on the coordinates field shown above.

4) "object" classification loss. For a positive sample, the classification loss is specified as cross entropy loss,

$$loss_{class} = -\lambda_{class} \sum_{i=0}^{s\times s} \sum_{j=0}^{3} 1_{ij}^{obj} \log\left(\frac{exp(d_c)}{\sum_{k=1}^{C} exp(d_k)}\right) \tag{10.12}$$



Note that $d_k, k = 1,2,\dots,C$, are the unnormalized logits from the last Conv2d layer for the grid scale prediction in YOLO v3 model. Thus, in PyTorch we can call *nn.CrossEntropyLoss*() to calculate $loss_{class}$, because *nn.CrossEntropyLoss*() combines the nn.LogSoftmax and nn.NLLLoss functions to compute the loss.

The total loss at the grid *s* is the sum of all above loss components (10.9), (10.10), (10.11), (10.12) as

$$\begin{aligned} Loss(s) &= loss_{noobj} + loss_{obj} + loss_{box} + loss_{class} \\ &= \lambda_{coord} \sum_{i=0}^{s\times s} \sum_{j=0}^{3} 1_{ij}^{obj} \left[\left(\hat{x} - \sigma(t_x)\right)^2 + \left(\hat{y} - \sigma(t_y)\right)^2 + (\hat{t}_w - t_w)^2 + (\hat{t}_h - t_h)^2\right] \\ &\quad -\lambda_{noobj} \sum_{i=0}^{s\times s} \sum_{j=0}^{3} 1_{ij}^{noobj} \log\left(1 - \sigma(t_o)\right) \\ &\quad + \lambda_{obj} \sum_{i=0}^{s\times s} \sum_{j=0}^{3} 1_{ij}^{obj} \left(IOU - \sigma(t_o)\right)^2 \\ &\quad -\lambda_{class} \sum_{i=0}^{s\times s} \sum_{j=0}^{3} 1_{ij}^{obj} \log\left(\frac{exp(d_c)}{\sum_{k=1}^{C} exp(d_k)}\right) \end{aligned} \tag{10.13}$$

where $s = 13, 26, 52$.

The final loss function of YOLO v3 model is the sum of losses at three grid scales,

$$LOSS = Loss(13) + Loss(26) + Loss(52) \tag{10.14}$$

Note that λ_{noobj}, $\lambda_{obj}, \lambda_{coord}, \lambda_{class}$ are the weights for the loss components, which balance the contribution of each loss component to the final loss. For example, the number of negative

samples in the target is much larger than that of positive samples, λ_{noobj} is set to a smaller number (e.g., 0.5) than the value of λ_{obj} (e.g., 5). The mask element, for grid cell *i* and anchor box *j*, is defined as

$$1_{ij}^{obj} = \begin{cases} 1 & positive\ sample\ (i.e., confidence = 1) \\ 0 & negative\ sampe (i.e., confidence = 0) \end{cases} \tag{10.15}$$

$$1_{ij}^{noobj} = \begin{cases} 0 & positive\ sample\ (i.e., confidence = 1) \\ 1 & negative\ sampe (i.e., confidence = 0) \end{cases} \tag{10.16}$$

10.5 Implementation of YOLO v3 Using Pre-trained Model

This section gives a comprehensive tutorial of YOLO v3 implementation based on the pre-trained model by Joseph Redmon (the inventor of YOLO). Three files are downloaded for this section: yolov3.cfg, yolov3.weights, and coco.names.

yolov3.cfg: https://github.com/pjreddie/darknet/blob/master/cfg/yolov3.cfg, a configuration file that defines the structure of YOLO v3 model.
yolov3.weights: https://pjreddie.com/media/files/yolov3.weights, a binary file that stores the weights in a float data type.
coco.names: https://github.com/pjreddie/darknet/tree/master/data, a text file that lists the name of classes in COCO dataset.

Fig.10.16 illustrates the flowchart of YOLO v3 implementation. The configuration file, *yolov3.cfg*, defines the architecture of YOLO v3 model. We use a function *parse_cfg*() to parse the cfg file, and eventually generate *yolonet*() as an nn.Module. The model *yolonet*() takes images as its input, and delivers all bounding boxes for all grid cells. The function *non_max_suppression*() generates the final detected bounding boxes by eliminating the low confident bounding boxes and redundant bounding boxes via the non-max suppression algorithm. We will explore the details of the implementation in the following subsections.

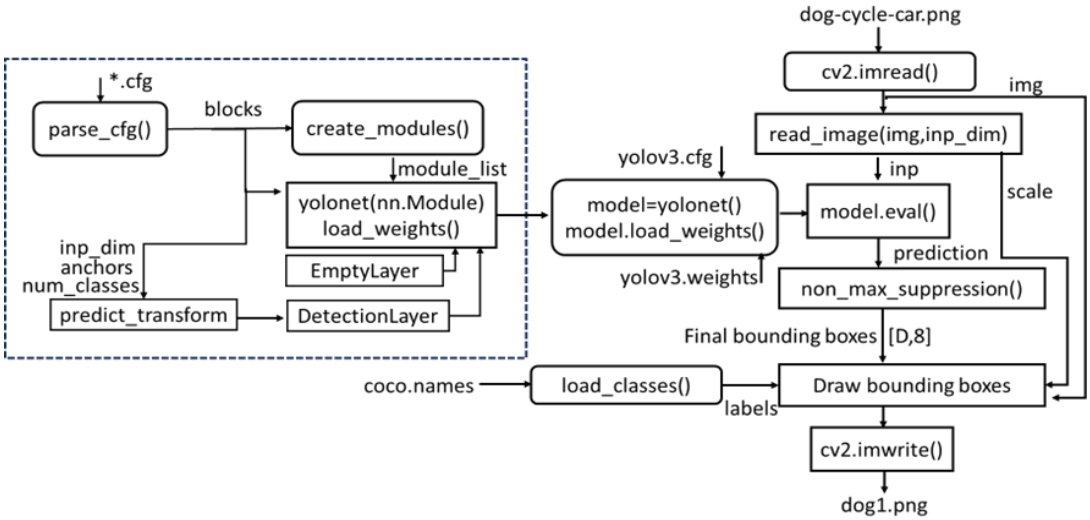


Fig.10.16 Flowchart of YOLO v3 implementation

10.5.2 Model architecture specified by a configuration file: yolov3.cfg

It is common practice to specify the architecture of a deep neural network by a configuration (or config) file, and then convert it to a neural network module described by a standard framework (e.g. *nn.Module* in PyTorch). A config file (*.cfg) is a text file that consists of a sequence of blocks.

In yolov3.cfg, there are six types of blocks: **[net], [convolutional], [shortcut], [route], [upsample]** and **[yolo]**. Each block, except the first block [net], specifies a layer. The first block in yolov3.cfg, called **[net]**, describes the information on the network input and training/testing parameters. Each of the subsequent blocks belongs to one of the other five types: [convolutional], [shortcut], [route], [upsample], [yolo], and specifies a layer. A block **[convolutional]** defines the structure of a conv layer, including the number of filters, filter size, stride, zero-padding, batch normalization and activation function. The number of input channels for a layer is defined by the number of output channels in the previous layer. A **[shourtcut]** block specifies a jump connection. A **[route]** block has an attribute "*layers"* which can have either one, or two values. When *layers* attribute has only one value, it outputs the feature maps of the layer indexed by the value. When *layers* has two values, it returns the concatenated feature maps of the layers indexed by the two values. A block **[upsample]** up-samples the feature map from the previous layer by a factor of stride. A **[yolo]** block provides the parameters required for detection, and it corresponds to the detection layer, which processes the output feature map tensor for detection.

As examples, Table 10.3 lists some blocks in yolov3.cfg. The **[convolutional]** block specifies a convolutional layer with batch normalization and leakyReLU activation. The **[shourtcut]** block describes a jump connection *from* layer -3, which means the output of the shortcut layer is obtained by adding feature maps from the previous (i.e. the first layer backwards) and the 3rd layer backwards from the shortcut layer. The block "**[route]** layers = -4" specifies a layer that outputs the feature map from the 4th layer backwards. The block "**[route]** layers = -1, 61" specifies a layer that outputs a concatenated (along with depth dimension) feature map from the previous layer (-1) and the 61st layer. The **[upsample]** block defines a layer that up-samples the feature map from the previous layer by a factor of stride 2 using a default mode – nearest mode. The **[yolo]** block gives anchor boxes and other parameters for detection at the corresponding scale. The *anchors* give all anchors while *mask* specifies the corresponding three anchors for this grid scale.

Table 10.3 Examples of blocks in yolov3.cfg

<table>
<tr><td>[convolutional]

batch_normalize=1
filters=32
size=3
stride=1
pad=1
activation=leaky</td><td colspan="3">[yolo]
mask = 6,7,8
anchors = 10,13, 16,30, 33,23, 30,61, 62,45, 59,119, 116,90, 156,198, 373,326
classes=80
num=9
jitter=.3
ignore_thresh = .5
truth_thresh = 1
random=1</td></tr>
<tr><td>[shortcut]
from=-3
activation=linear</td><td>[route]
layers = -4</td><td>[route]
layers = -1, 61</td><td>[upsample]
stride=2</td></tr>
</table>

10.5.2 Create the model and load the weights

In this section, we will explore how to create the neural network model and load the pre-trained weights to the model, from two files – yolov3.cfg and yolov3.weights. All Python codes can be run through Jupyter notebook.

We start by importing the basic packages.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable
import numpy as np
import cv2
```

The function *parse_cfg* is to parse the cfg file and store every block as a dictionary data type. The attributes of each block and their values are stored as key-value pairs in the dictionary. As we parse through the cfg, we keep appending these dictionaries, denoted by the variable *block* in the code, to a list *blocks*. The function returns this list *blocks*.

```
def parse_cfg(cfgfile):
    """
    input: a configuration file, eg. yolov3.cfg

    Returns: blocks, which is a list of blocks.
        -- each block corresponds to a block in cfg file
        -- each block is represented as a dictionary
    """

    file = open(cfgfile, 'r')
    #lines = file.read().split('\n')                        # store the lines in a list
    lines = file.read().splitlines()                        # store the lines in a list
    lines = [x for x in lines if len(x) > 0]                # remove the empty lines
    lines = [x for x in lines if x[0] != '#']               # remove the comment lines
    lines = [x.rstrip().lstrip() for x in lines]
    # remove leading and trailing whitespaces
    #print(lines)
    # lines: a list of strings, each string is an effective line in cfg

    block = {}           # inital a dict
    blocks = []          # initial list

    for line in lines:
        if line[0] == "[":
        # line is a string, line[0] is the first character in the line
            if len(block) != 0:
                # If block is not empty,
                # the "block" is for the previous block
                # and ready to store to "blocks"
                blocks.append(block)      # add it the blocks list
                block = {}                # re-init the block
            block["type"] = line[1:-1].rstrip()
          # store the block "type" for current block
        else:
```

```
            key,value = line.split("=")
            block[key.rstrip()] = value.lstrip() #store "value" to block[key]
    blocks.append(block)            # store the last "block" to the list

  return blocks
```

The content of "blocks" is printed below. The "blocks" include 108 blocks. The first block is "net" about the information of the model, The remaining 107 blocks correspond to 107 layers in Fig.10.12.

```
print(blocks)
```

```
[{'type': 'net', 'batch': '1', 'subdivisions': '1', 'width': '416', 'height':
'416', 'channels': '3', 'momentum': '0.9', 'decay': '0.0005', 'angle': '0',
'saturation': '1.5', 'exposure': '1.5', 'hue': '.1', 'learning_rate':
'0.001', 'burn_in': '1000', 'max_batches': '500200', 'policy': 'steps',
'steps': '400000,450000', 'scales': '.1,.1'}, {'type': 'convolutional',
'batch_normalize': '1', 'filters': '32', 'size': '3', 'stride': '1', 'pad':
'1', 'activation': 'leaky'},…]
```

The *create_modules* function takes the list *blocks* generated by the *parse_cfg* function, and returns *net_info* and *module_list*. The *net_info* is a dictionary that stores information about the network. The *module_list* is a list nn.ModuleList() that contains all 107 layers in YOLO v3, including the index of each layer. The *EmptyLayer()* is defined for route and shortcut layers while *DetectionLayer()* is defined for the YOLO detection head. Each element in *module_list* corresponds to a layer in Fig.10.12.

```
class EmptyLayer(nn.Module):
    def __init__(self):
        super(EmptyLayer, self).__init__()


class DetectionLayer(nn.Module):
    def __init__(self, anchors):
        super(DetectionLayer, self).__init__()
        self.anchors = anchors

def create_modules(blocks):
    """
    input: a list -- blocks, each element is a dict -- block
    returns: net_info -- the first block about the net
             module_list: a list of nn.modules
    """
    net_info = blocks[0]         # net info is a dict for the net information
    module_list = nn.ModuleList()  # init a list containing nn.modules
    prev_filters = 3               # initial prev_filters
    output_filters = []            # store

    for index, x in enumerate(blocks[1:]):
        module = nn.Sequential()    # initial module for current block

        #start with blocks[1]
```

```
        #check the type of block
        #create a new module for the block
        #append to module_list

        #If a convolutional layer
        if (x["type"] == "convolutional"):
            #Get the info about the layer
            activation = x["activation"]
            try:
                batch_normalize = int(x["batch_normalize"])
                bias = False
            except:
                batch_normalize = 0
                bias = True

            filters= int(x["filters"])
            padding = int(x["pad"])          # whether zero padding
            kernel_size = int(x["size"])
            stride = int(x["stride"])

            if padding:
                pad = (kernel_size - 1) // 2 #pad:the zero pad in nn.Conv2d()
            else:
                pad = 0

            #Add the convolutional layer
            conv = nn.Conv2d(prev_filters, filters, kernel_size, stride, pad,
bias = bias)
            module.add_module("conv_{0}".format(index), conv)

            #Add the Batch Norm Layer
            if batch_normalize:
                bn = nn.BatchNorm2d(filters)
                module.add_module("batch_norm_{0}".format(index), bn)

            #Check the activation.
            if activation == "leaky":
                activn = nn.LeakyReLU(0.1, inplace = True)
                module.add_module("leaky_{0}".format(index), activn)

        # If an upsampling layer
        # nearest mode, factor = stride
        elif (x["type"] == "upsample"):
            stride = int(x["stride"])
            upsample = nn.Upsample(scale_factor = stride, mode = "nearest")
            module.add_module("upsample_{}".format(index), upsample)
            # filters is the same as the previous layer, no need to update

        #If a route layer
        elif (x["type"] == "route"):

            route = EmptyLayer()
            module.add_module("route_{0}".format(index), route)

            x["layers"] = x["layers"].split(',')
            #Start  of a route
            start = int(x["layers"][0])    # the first integer e.g -1, or -4
```

```
            #end, if there is a second value for layers, e.g, 36, or 61.
            try:
                end = int(x["layers"][1])  # second value
            except:
                end = 0    # no second value

            if start > 0:
                start = start - index
            if end > 0:
                end = end - index

            if end == 0:   # only one value (start)
                filters = output_filters[index + start]
            else:  # two values (start, end)
                filters=output_filters[index+start]+output_filters[index+end]


        #shortcut corresponds to skip connection
        elif x["type"] == "shortcut":
            shortcut = EmptyLayer()
            module.add_module("shortcut_{}".format(index), shortcut)
            # no need to update filters

        #Yolo is the detection layer
        elif x["type"] == "yolo":
            mask = x["mask"].split(",")
            mask = [int(x) for x in mask]

            anchors = x["anchors"].split(",")
            anchors = [int(a) for a in anchors]
            anchors=[(anchors[i],anchors[i+1]) for i in
range(0,len(anchors),2)]
            anchors = [anchors[i] for i in mask]

            detection = DetectionLayer(anchors)
            module.add_module("Detection_{}".format(index), detection)
            # no need to update filters

        module_list.append(module)      # add module to the module_list
        prev_filters = filters
        # will be used for the input channels if the next layer is conv2d

        output_filters.append(filters)
        # save the channels for layers to be used in layer "route"

    return (net_info, module_list)
```

module_list can be printed as below, and its length is 107.

```
print(create_modules(blocks))
```

```
({'type': 'net', 'batch': '1', 'subdivisions': '1', 'width': '416', 'height':
'416', 'channels': '3', 'momentum': '0.9', 'decay': '0.0005', 'angle': '0',
'saturation': '1.5', 'exposure': '1.5', 'hue': '.1', 'learning_rate':
```

```
'0.001', 'burn_in': '1000', 'max_batches': '500200', 'policy': 'steps',
'steps': '400000,450000', 'scales': '.1,.1'}, ModuleList(
  (0): Sequential(
    (conv_0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1), bias=False)
    (batch_norm_0): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (leaky_0): LeakyReLU(negative_slope=0.1, inplace=True)
  )
  (1): Sequential(
    (conv_1): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1,
1), bias=False)
    (batch_norm_1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (leaky_1): LeakyReLU(negative_slope=0.1, inplace=True)
  )
…)
```

Based on the functions *parse_cfg()* and *create_modules()*, we can build the entire neural network for YOLO v3, called *Yolonet()*. In addition to the basic layers in PyTorch, such as *nn.Con2d()*, *nn.BatchNorm2d()*, *nn.LeakyReLU()*, an important function, called *predict_transform()*, has been defined for the operation in *DetectionLayer()*, which is illustrated in Fig.10.17.

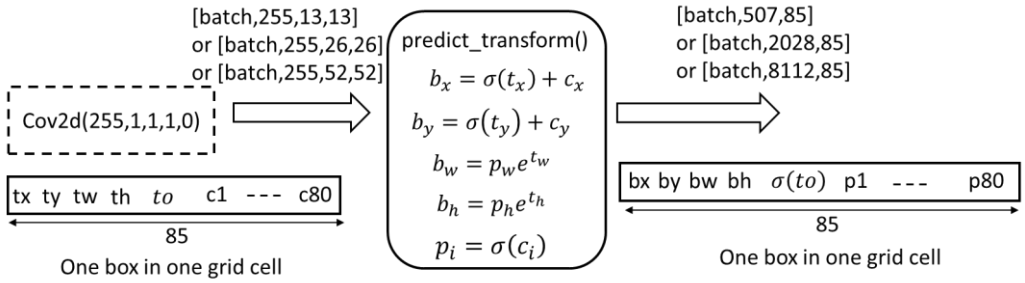


Fig.10.17 the operation of *predict_transform*(), c_x, c_y are the coordinates of the grid cell.

```
def predict_transform(prediction, inp_dim, anchors, num_classes):
    # this function is used in yolo DetectionLayer()
    # generates all predictions for one grid scale
    #   1) re-organize the tensor
    #   2) scale the anchors
    #   3) coordinate transform
    #   4) apply sigmoid for class probabilities

    """
    inputs:
        -- prediction: output from the last conv2d
           shape [batch, 255, grid_size, grid_size] (e.g.[b, 255,13,13])
        -- inp_dim: input size, integer, e.g. 416
        -- anchors: a list of 3 anchors for this grid scale,
                    [(*,*), (*,*), (*,*)]
        -- num_classes: the number of classes, integer, e.g. 80
    returns:
        -- prediction: tensor shape [batch, grid_size x grid_size x 3, 85]
      each box: prediction[i,j,:] =(bx,by,bw,bh,sigmoid(to), p1, ..., p80)
```

```
    """

    batch_size = prediction.size(0)
    stride =  inp_dim // prediction.size(2)
    grid_size = prediction.size(2)
    bbox_attrs = 5 + num_classes
    num_anchors = len(anchors)

    prediction = prediction.view(batch_size, bbox_attrs*num_anchors,
grid_size*grid_size)
    #example: [b, 255,13,13]--> [b, 255, 169]

    prediction = prediction.transpose(1,2).contiguous()
    #example: [b,255,169] --> [b,169,255]

    prediction = prediction.view(batch_size, grid_size*grid_size*num_anchors,
bbox_attrs)
    #example: [b,169,255] --> [b,169x3,85]=[b,507,85]

    anchors = [(a[0]/stride, a[1]/stride) for a in anchors]
    # scaled to unit grid size

    #Sigmoid the  centre_X, centre_Y. and object confidencce
    prediction[:,:,0] = torch.sigmoid(prediction[:,:,0])
    prediction[:,:,1] = torch.sigmoid(prediction[:,:,1])
    prediction[:,:,4] = torch.sigmoid(prediction[:,:,4])

    #Add the center offsets
    grid = np.arange(grid_size)
    a,b = np.meshgrid(grid, grid)    # a: (13,13), b: (13,13)

    x_offset = torch.FloatTensor(a).view(-1,1)  #shape(169,1)
    y_offset = torch.FloatTensor(b).view(-1,1)  #shape(169,1)

    x_y_offset = torch.cat((x_offset, y_offset),
1).repeat(1,num_anchors).view(-1,2).unsqueeze(0)
    # x_y_offset, shape (1, 507, 2), grid coordinates repeated 3 time.
    # [[0.,0.], [0., 0.], [0.,0.],
    #  [1.,0.], [1.,0.], [1.,0.],
    #  [2.,0.], [2.,0.], [2.,0.],
    #  ....
    #  [12.,12.], [12.,12.], [12.,12.]]

    prediction[:,:,:2] += x_y_offset
    # bounding box: bx, by

    #log space transform height and the width
    anchors = torch.FloatTensor(anchors)


    anchors = anchors.repeat(grid_size*grid_size, 1).unsqueeze(0)
    # shape: (1, 507, 2) to match prediction (1, 507, 85)

    prediction[:,:,2:4] = torch.exp(prediction[:,:,2:4])*anchors
    # bounding box: bw, bh
```

```
    prediction[:,:,5: 5 + num_classes] = torch.sigmoid((prediction[:,:, 5 : 5
+ num_classes]))

    prediction[:,:,:4] *= stride
    # scaled back to unit of pixel, shape is [batch, 13x13x3, 85]
    # one bounding box prediction[1,1,:] is (bx,by,bw,bh,sigmoid(to), p1, p2,
..., p80)

    return prediction
```

In class *Yolonet*(), we construct the forward path of YOLO v3, and also define `load_weights(self, weightfile)` to add an attribute to *Yolonet*() for loading the pre-trained weights.

```
class Yolonet(nn.Module):
    def __init__(self, cfgfile):
        super(Yolonet, self).__init__()
        self.blocks = parse_cfg(cfgfile)
        self.net_info, self.module_list = create_modules(self.blocks)

    def forward(self, x):
        modules = self.blocks[1:]
        outputs = {}   # We save the outputs for the route or shortcut layer,
                       # key is layer index, value is the output of the layer

        first = 1
        for i, module in enumerate(modules):
            module_type = (module["type"])

            if module_type == "convolutional" or module_type == "upsample":
                x = self.module_list[i](x)

            elif module_type == "route":
                layers = module["layers"]
                layers = [int(a) for a in layers]

                if (layers[0]) > 0:
                    layers[0] = layers[0] - i

                if len(layers) == 1:
                    x = outputs[i + (layers[0])]

                else:
                    if (layers[1]) > 0:
                        layers[1] = layers[1] - i

                    map1 = outputs[i + layers[0]]
                    map2 = outputs[i + layers[1]]
                    x = torch.cat((map1, map2), 1)

            elif  module_type == "shortcut":
                from_ = int(module["from"])
                x = outputs[i-1] + outputs[i+from_]

            elif module_type == 'yolo':
```

```
                anchors = self.module_list[i][0].anchors
                #Get the input dimensions
                inp_dim = int (self.net_info["height"])

                #Get the number of classes
                num_classes = int (module["classes"])

                #Transform
                x = x.data
                x = predict_transform(x, inp_dim, anchors, num_classes)
                if first:               #if the first yolo detectio head.
                    detections = x
                    first = 0

                else:       # if the 2nd and 3rd head
                    detections = torch.cat((detections, x), 1)

            outputs[i] = x

        return detections


    def load_weights(self, weightfile):
        # Open weightfile
        fp = open(weightfile, "rb")

        #The first 5 values are header information
        header = np.fromfile(fp, dtype = np.int32, count = 5)
        self.header = torch.from_numpy(header)
        self.seen = self.header[3]

        weights = np.fromfile(fp, dtype = np.float32)

        ptr = 0
        for i in range(len(self.module_list)):
            module_type = self.blocks[i + 1]["type"]

            # If module_type is convolutional load weights
            # Otherwise do nothing.

            if module_type == "convolutional":
                model = self.module_list[i]
                try:
                    batch_normalize=int(self.blocks[i+1]["batch_normalize"])
                except:
                    batch_normalize = 0

                conv = model[0]


                if (batch_normalize):   # if BN applied, load BN parameters
                    bn = model[1]

                    #Get the number of weights of Batch Norm Layer
                    num_bn_biases = bn.bias.numel()

                    #Load the weights
```

```
                    bn_biases = torch.from_numpy(weights[ptr:ptr +
num_bn_biases])
                    ptr += num_bn_biases

                    bn_weights = torch.from_numpy(weights[ptr: ptr +
num_bn_biases])
                    ptr  += num_bn_biases

                    bn_running_mean = torch.from_numpy(weights[ptr: ptr +
num_bn_biases])
                    ptr  += num_bn_biases

                    bn_running_var = torch.from_numpy(weights[ptr: ptr +
num_bn_biases])
                    ptr  += num_bn_biases

                    #Cast the loaded weights into dims of model weights.
                    bn_biases = bn_biases.view_as(bn.bias.data)
                    bn_weights = bn_weights.view_as(bn.weight.data)
                    bn_running_mean =bn_running_mean.view_as(bn.running_mean)
                    bn_running_var = bn_running_var.view_as(bn.running_var)

                    #Copy the data to model
                    bn.bias.data.copy_(bn_biases)
                    bn.weight.data.copy_(bn_weights)
                    bn.running_mean.copy_(bn_running_mean)
                    bn.running_var.copy_(bn_running_var)

                else: # if no BN, load conv layer biases
                    #Number of biases
                    num_biases = conv.bias.numel()

                    #Load the weights
                    conv_biases = torch.from_numpy(weights[ptr: ptr +
num_biases])
                    ptr = ptr + num_biases

                    #reshape the loaded weights according to the dims of the
model weights
                    conv_biases = conv_biases.view_as(conv.bias.data)

                    #Finally copy the data
                    conv.bias.data.copy_(conv_biases)

                #load the weights for the Convolutional layers
                num_weights = conv.weight.numel()
                conv_weights = torch.from_numpy(weights[ptr:ptr+num_weights])
                ptr = ptr + num_weights

                conv_weights = conv_weights.view_as(conv.weight.data)
                conv.weight.data.copy_(conv_weights)
```

We can instantiate *Yolonet()* and load the weights as below.

```
model = Yolonet("cfg/yolov3.cfg")
model.load_weights("yolov3.weights")
```

Given a batch of images, to get the output from model, we just run: *pred = model(inp)*. The input, *inp*, is a tensor with shape [Batch_size, Channel, Height, Width] assuming all images have the same Height and Width. The output, *pred*, is a tensor that contains the predicted bounding boxes for all images, and its shape is [Batch_size, number of boxes, 85], where

$$number\ of\ boxes = (13 \times 13 + 26 \times 26 + 52 \times 52) \times 3 = 10647$$

when image size is $(3 \times 416 \times 416)$ and the number of classes is 80. The prediction for image *i* is *pred[i,:,:]*, shown in Fig.10.18. There are three bounding boxes for one grid cell. The prediction for each bounding box includes 85 numbers: 4 for box center and width and height, 1 for object confidence score, 80 for class probabilities.

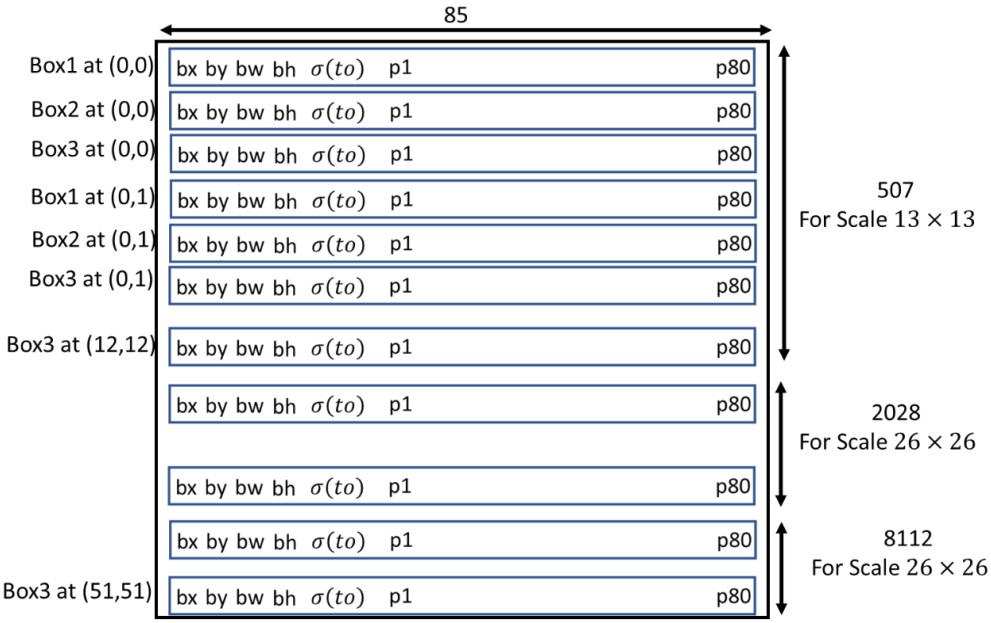


Fig.10.18 The output of *Yolonet* () for one image *i*, *pred* [i,:,:]. Note that the unit of bx,by,bw,bh is a pixel, and that p1,…, p80 are class probabilities after sigmoid function.

```
inp_random = torch.rand(2,3,416,416)
pred_random = model(inp_random)
pred_random.shape
```

```
torch.Size([2, 10647, 85])
```

10.5.3 Non-max suppression

The model *Yoloknet* delivers all bounding boxes for a batch of images, with 10647 bounding boxes per image, as shown in Fig.10.18. Since most of the bounding boxes with low confidence scores are not responsible for any object or some of them points to the same object, we need to eliminate the low confident bounding boxes (i.e., $\sigma(to) < threshold$) and the redundant bounding boxes for one object, so that each detected object is framed by only one bounding box. This is done by the algorithm of non-max suppression.

First, we define a function, bbox_iou(box1,box2), to calculate the intersection over union between box1 and box2. Then the function, *non_max_suppression(prediction, confidence, num_classes,*

nms_conf), is defined to convert the output of *Yolonet*, *prediction*, to the finally detected bounding boxes, *output*, through non-max suppression. The tensor *prediction* has a shape of [Batch_size, 10647, 85] while the tensor *output* has a shape of [D,8], where D is the total number of detected bounding boxes for the batch. The format of each bounding box in prediction or output is illustrated in Fig.10.19.

Specifically, *non_max_suppresion* () performs the following operations in order: 1) zero out all bounding boxes whose object confidence score $\sigma(to)$ is less than the threshold *confidence*; 2) convert center coordinates to corner coordinates for each bounding box; 3) on each image, perform non-max suppression class-wisely; 4) attach the image index to the corresponding bounding boxes, and keep the maximum of class probabilities and the corresponding class index, while discarding other class probabilities. As the result, each detected bounding box has a format of an 8-element vector: (image_index, x1, y1, x2, y2, $\sigma(to)$, max(pc), argmax(pc)).

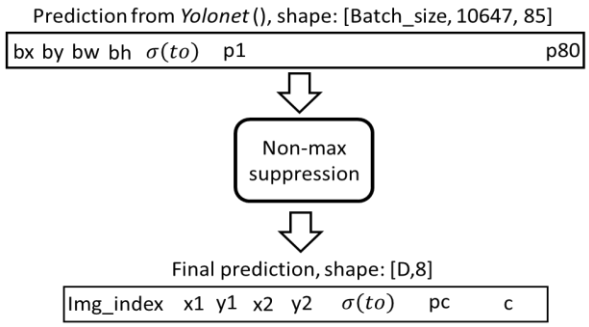


Fig.10.19 Input and output of non-max suppression. D is the total number of final bounding boxes in the batch.

```
def bbox_iou(box1, box2):
    """
    inputs:
        --box1: tensor shape [N1,4], given as top-left and bottom-right
        --box2: tensor shape [N2,4], given as top-left and bottom-right
            box1 and box2 should include the same number of boxes (N1=N2),
         or one of them includes only one box and the other includes
            multiple boxes (N1=1, or N2=1)

    Returns
        -- if N1 =N2, returns one-to-one IOUs
        -- if N1=1 or N2=1, returns one-to-many IOUs

    """
    #Get the coordinates of bounding boxes
    b1_x1, b1_y1, b1_x2, b1_y2 = box1[:,0], box1[:,1], box1[:,2], box1[:,3]
    b2_x1, b2_y1, b2_x2, b2_y2 = box2[:,0], box2[:,1], box2[:,2], box2[:,3]

    #get the corrdinates of the intersection rectangle
    inter_rect_x1 =  torch.max(b1_x1, b2_x1)
    inter_rect_y1 =  torch.max(b1_y1, b2_y1)
    inter_rect_x2 =  torch.min(b1_x2, b2_x2)
    inter_rect_y2 =  torch.min(b1_y2, b2_y2)
```

```
    #Intersection
    inter_area = torch.clamp(inter_rect_x2 - inter_rect_x1 + 1, min=0) *
torch.clamp(inter_rect_y2 - inter_rect_y1 + 1, min=0)

    #Union
    b1_area = (b1_x2 - b1_x1 + 1)*(b1_y2 - b1_y1 + 1)
    b2_area = (b2_x2 - b2_x1 + 1)*(b2_y2 - b2_y1 + 1)

    #IOU
    iou = inter_area / (b1_area + b2_area - inter_area)

    return iou
```

```
def non_max_suppression(prediction, confidence, num_classes, nms_conf = 0.4):
    """
    delivers all final bounding boxes, which are ready to draw on original
    images.
inputs:
 -- prediction, from predict_transform() in Darkent model, includes all
     bounding boxes shape is [batch, (13x13+26x26+52x52)x3, 85]
      =[batch, 10647, 85], 10647 bounding boxes per batch
      prediction[0,0,:]:
      bx,by,bw,bh,sigmoid(t0),p1=sigmoid(), p2=sigmoid()..., p80=sigmoid()
 -- confidence, object confidence threshold (e.g. 0.5) for sigmoid(t0) in
    each bounding box
 -- num_classes, integer, the number of class (e.g. 80)
 -- nms_conf, the threshold of IOU for non-max suppression
    outputs:
 -- output, a tensor [N, 8], N is the total number of predicted boxes for
    the batch
    each box:
    (image_index, x1,y1,x2,y2,sigmoid(to), class probability, class index)
    """

    conf_mask = (prediction[:,:,4] > confidence).float().unsqueeze(2)
    #.unsqueeze(2) keep the dimension 2 to match the prediction dimension
    #[batch, 10647,85]
    # conf_mask shape is [batch, 10647,1]

    prediction = prediction*conf_mask
    # clear all low confident bounding boxes to zero-vectors
    # maintain all other bounding boxes

    # convert (center,w,h) format to (top-left, bottom-right) format
    box_corner = prediction.new(prediction.shape)  #create a new tensor
    box_corner[:,:,0] = (prediction[:,:,0] - prediction[:,:,2]/2)
    box_corner[:,:,1] = (prediction[:,:,1] - prediction[:,:,3]/2)
    box_corner[:,:,2] = (prediction[:,:,0] + prediction[:,:,2]/2)
    box_corner[:,:,3] = (prediction[:,:,1] + prediction[:,:,3]/2)
    prediction[:,:,:4] = box_corner[:,:,:4]
    # now, prediction in a corner format:
    #[x1, y1, x1, y2, sigmoid(t0), sigmoid().....]


    batch_size = prediction.size(0)
```

```
    #initial
    first = True


    for ind in range(batch_size):
        image_pred = prediction[ind]
       # image_pred [10647,85]: prediction for one image
       #confidence threshholding
       #NMS

        max_conf, max_conf_score=torch.max(image_pred[:,5:5+num_classes], 1)
        max_conf = max_conf.float().unsqueeze(1)
        max_conf_score = max_conf_score.float().unsqueeze(1)
        # max_conf: max value, shape [10647,1]
        # max_conf_score: max index, shape [10647,1]

        seq = (image_pred[:,:5], max_conf, max_conf_score)
        image_pred = torch.cat(seq, 1)
        #shape [10647,7], (x1,y1,x2,y2,sigmoid(to),pc_max,class_max)

        non_zero_ind =  (torch.nonzero(image_pred[:,4]))
        try:
            image_pred_ = image_pred[non_zero_ind.squeeze(),:].view(-1,7)
        except:
            continue

        if image_pred_.shape[0] == 0:
            continue
        # image_pred_, includes only detected bounding boxes, shape [M1, 7]

        #Get the various classes detected in the image
        img_classes = torch.unique(image_pred_[:,-1])
        # keep one element for each detected class.
        # img_class shape [M2], M2 < or = M1

        for cls in img_classes:
            #perform NMS for each class

            #get the detections with one particular class
            cls_mask=image_pred_*(image_pred_[:,-
1]==cls).float().unsqueeze(1)
            class_mask_ind = torch.nonzero(cls_mask[:,-2]).squeeze()
            image_pred_class = image_pred_[class_mask_ind].view(-1,7)
            # image_pred_class [M3,7]: detected bounding boxes for class cls

            # sort the detections such that the bounding box
            # with the maximum objectness confidence is at the beginning
            conf_sort_index=torch.sort(image_pred_class[:,4],descending=True
)[1]
            image_pred_class = image_pred_class[conf_sort_index]
            idx = image_pred_class.size(0)
            #Number of detected bounding boxes for class cls

            for i in range(idx):
                #Get the IOUs of all boxes that come after the one we are loo
                # king at in the loop
                try:
```

```
                    ious = bbox_iou(image_pred_class[i].unsqueeze(0),
image_pred_class[i+1:])
                    # compute IOUs between box[i] and all the rest boxes
                except ValueError:
                    break

                except IndexError:
                    break

                #Zero out all the detections that have IoU > treshhold
                iou_mask = (ious < nms_conf).float().unsqueeze(1)
                image_pred_class[i+1:] *= iou_mask

                #Remove the bounding boxes whose IOU with box[i]
                #is greater than nms_conf
                non_zero_ind = torch.nonzero(image_pred_class[:,4]).squeeze()
                image_pred_class = image_pred_class[non_zero_ind].view(-1,7)

            batch_ind = image_pred_class.new(image_pred_class.size(0),
1).fill_(ind)

            seq = (batch_ind, image_pred_class)
            # attach the batch index to the bounding boxes for class cls

            if first:
                output = torch.cat(seq,1)
                first = False
            else:
                out = torch.cat(seq,1)
                output = torch.cat((output,out))
            # output shape [M4, 8], M4 is the number of valid bounding boxes
            # up to current batch, image, class cls

        # output shape [M5, 8], M5 is the number of valid BBs up to
        # current image

    try:
        return output
    except:
        return 0
    # the returned output [D,8], includes D bounding boxes for the batch
    # each box:
    # (image_index, x1,y1,x2,y2,sigmoid(to), class probability, class index)
```

10.5.4 Put all together

Finally, we apply the model *Yolonet* with pre-trained weights and non-max suppression for an object detection task. The final detected bounding boxes are plotted on the testing image.

Since the pre-trained model was trained on COCO dataset, a list of class names in the dataset is created, and saved as one line per name in a text file, named *coco.names*. We define a function, *load_classes* (), to read the names from the file, and return a list of classes. With this list, we can display the name of detected object in the image.

```
def load_classes(namesfile):
```

```
    fp = open(namesfile, "r")
    names = fp.read().split("\n")[:-1]
    return names
```

We use CV2 to load the images. Since CV2 loads an image as a numpy array [H,W,Channel], with BGR as the order of the color channels, and PyTorch neural network input format is [Batch, Channel, inp_dim, inp_dim], with the channel order of RGB, we need to write a function *read_image* to transform the numpy array into PyTorch input format. First, *read_image*() scales the original image ($H \times W$) to fit the frame ($inp\_dim \times inp\_dim$) as large as possible while keeping the aspect ratio unchanged, and pads the left out areas with the color (128,128,128), illustrated in Fig.10.20. Then *read_image*() convert the data format to [Batch, Channel, inp_dim, inp_dim]. The scale factor is also returned to be used for drawing the detected bounding boxes on the original image.

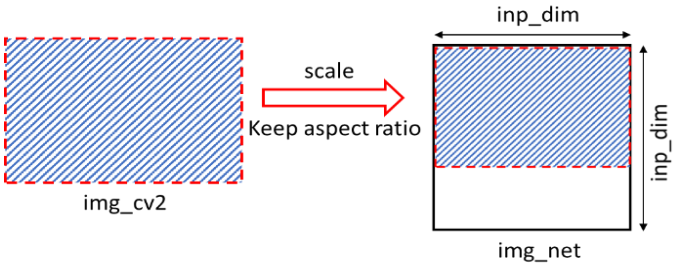


Fig.10.20 Scale the image to fit the input size $inp\_dim \times inp\_dim$

```
def read_image(img_cv2, inp_dim):
    """
    converts and scale the image read by cv2 to
    a tensor [batch,channel,inp_dim,inp_dim] for yolonet

    inputs:
        -- img_cv2: shape is [h,w,channel]: the orignal image from
cv2.imread
        -- inp_dim: integer, input size to yolo nueral network, e.g.
416
    Returns
        -- img_net: tensor shape is [batch, channel, height, width]
        -- scale_factor
    """
    #img, scale_factor = (scale_image(img, (inp_dim, inp_dim)))

    img_cv2_h, img_cv2_w = img_cv2.shape[0], img_cv2.shape[1]
    w = inp_dim
    h = inp_dim
    scale_factor = min(w/img_cv2_w, h/img_cv2_h)
    new_w = int(img_cv2_w * scale_factor)
    new_h = int(img_cv2_h * scale_factor)
    resized_image = cv2.resize(img_cv2, (new_w,new_h), interpolation =
cv2.INTER_CUBIC)
    # cv2.resize(img, (width, height), interpolation =...)

    canvas = np.full((inp_dim, inp_dim, 3), 128)
```

```
    canvas[0:new_h, 0:new_w, :] = resized_image
    # align the resized image to the top-left of canvas

    img_net = canvas[:,:,::-1].transpose((2,0,1)).copy()
    # canvas[:,:,::-1] reverse the number in color dimension BGR--> RGB
    # In CV2,channel order is [blue, green, red],
    # but in PyTorch [red, green, blue]
    # transpose((2,0,1)): convert [H,W,C] to [C,H,W] for PyTorch

    img_net = torch.from_numpy(img_net).float().div(255.0).unsqueeze(0)
    # add batch dimension
    # img_net: [1, channel, height, width]
    return img_net, scale_factor
```

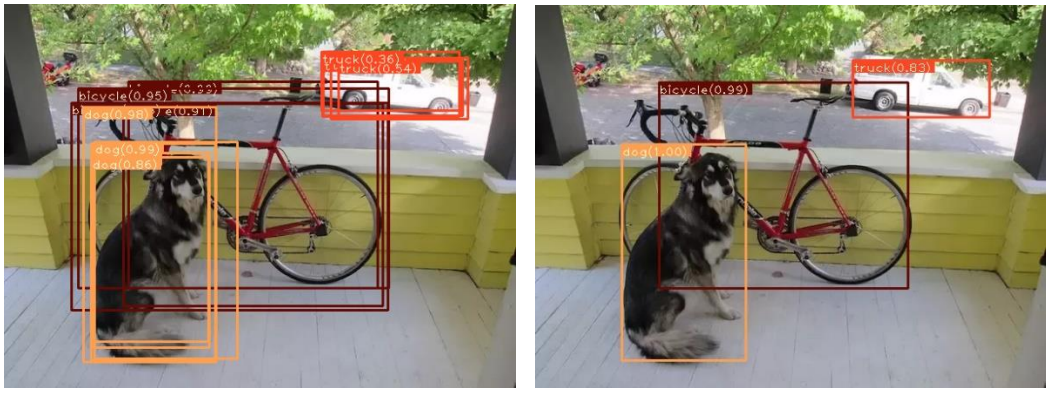
By putting all together, the following codes perform the object detection on the benchmark image "dog-cycle-car.png". If non-max-suppression is not applied (i.e., nms_thresh=1.0), there are multiple bounding boxes for one object, as shown in Fig.10.21(a). After non-max-suppression (e.g., nms_thresh=0.4) is applied, each detected object is frame by one bounding box, as shown in Fig.10.21(b).

```
import os

if not os.path.exists("./results"):
    os.makedirs("./results")

inp_dim = 416
batch_size = 1
confidence = 0.5 # threshold for confidence score, default 0.5
nms_thresh = 0.4 # threshold for nms iou, default 0.4
num_classes = 80
classes = load_classes("data/coco.names")

model = Yolonet("cfg/yolov3.cfg")
model.load_weights("yolov3.weights")
model.eval()

img = cv2.imread("dog-cycle-car.png")
inp, scale_factor = read_image(img, inp_dim)
pred = model(inp)
prediction = non_max_suppression(pred, confidence, num_classes,
nms_conf = nms_thresh)

prediction[:,1:5] /= scale_factor

for i in range(prediction.shape[0]):
    x = prediction[i]

    c1 = (x[1:3].int()).tolist()
    c2 = (x[3:5].int()).tolist()
    #img = results[int(x[0])]
    cls = int(x[-1])
    color = (5*cls,10*cls,100*cls)
```

```
    label = "{0}".format(classes[cls])
    cv2.rectangle(img, c1, c2,color, 2)
    t_size = cv2.getTextSize(label, cv2.FONT_HERSHEY_PLAIN, 1 , 1)[0]
    c2 = c1[0] + t_size[0] + 50, c1[1] + t_size[1] + 4
    cv2.rectangle(img, c1, c2,color, -1)
    prob="(%.2f)" % round(float(x[5]*x[6]), 2)
    cv2.putText(img, label+str(prob), (c1[0], c1[1] + t_size[1] + 4),
cv2.FONT_HERSHEY_PLAIN, 1, [225,255,255], 1);

cv2.imwrite("./results/dog1.png", img)
```

```
print(prediction.shape)
```

```
tensor([[0.0000,146.6964, 91.9886,439.0550,334.3987,0.9917,0.9982,1.0000],
        [0.0000,374.5365, 65.5042,535.1949,132.5022,0.9943,0.8332,7.0000],
        [0.0000,102.2761,164.2881,248.8867,419.3336,1.0000,0.9996,16.0000]])
```



(a) without nms (i.e., nms_thresh=1) (b) with nms (nms_thresh=0.4)

Fig.10.21 YOLO v3 results on an image.

10.6 A Metric for Object Detection: mAP

Mean average precision (mAP) is a popular metric to measure the performance of an object detection model.

10.6.1 Precision and recall in object detection

The definition of mAP is built upon the concepts of precision and recall. The metrics, *precision* and *recall*, are typically used to measure the performance of a classification model. Precision is defined as the ratio between the predicted true positives and the total predicted positives, and thus it indicates the purity of predicted positives. On the other hand, recall is defined as the ratio between the predicted true positives and the total true positives, and thus it tells us what percentage of positive examples are correctly classified as positive.

$$Precision = \frac{TP}{TP+FP} \tag{10.17}$$

$$Recall = \frac{TP}{TP+FN} \tag{10.18}$$

Since the objective of object detection is not only to correctly classify the object (or objects) in the image but to also find where in the image it is located, we need to consider both the class and location of the objects in formulating precision and recall.

The computation of the overall mAP is based on a test dataset and the predictions of the model on the dataset. The test dataset consists of multiple images along with the ground truth bounding boxes. The predictions of the model on each image are the predicted bounding boxes, each of which includes the box location, confidence score and object class.

In object detection, precision and recall are calculated class-wisely. Let us consider one class in an image. A predicted box is counted as a *true positive* only if it meets the following two requirements: 1) there exists a ground truth box, whose IOU with the predicted box is greater than a predefined IOU threshold; and 2) no other predicted box has a larger IOU with this ground truth box than this predicted box. Requirement 1 guarantees the location of predicted box is close enough to the ground truth box while requirement 2 makes sure that each object can be predicted by no more than one box. If a predicted box is not a true positive, then it is a *false positive*. The total number of *false negatives* (FN) is the difference between the total number of ground truth boxes and the total number of true positives (TP).

As an example, Fig.10.22 illustrates the predicted bounding boxes (dashed line) versus ground truth boxes (solid line) in an image for a particular class, given a confidence threshold. In the image there are 6 objects (GT1, GT2, …, GT6) to be detected, but the model delivers all predicted boxes (B1, B2, …, B8) whose confidence scores exceeds the confidence threshold (e.g., 0.3). With an IOU threshold 0.5 (a default value), four predicted boxes B1, B4, B5 and B7 are considered as true positive. The rest of the predicted boxes (B2, B3, B6, B8) are considered as false positive. Note that B6 is considered as a false positive because the GT4 has been matched by a better predicted box B5. In this image, we have TP=4, FP=4, and FN=2 (two objects GT2 and GT6 are not correctly detected), and thus the precision and recall are equal to 4/8 and 4/6.

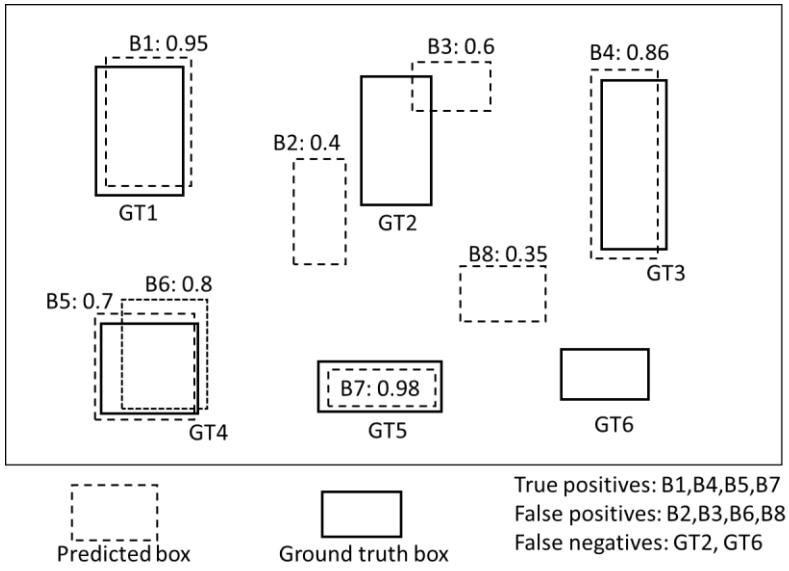


Fig.10.22 Examples of predicted boxes versus ground truth boxes in an image

10.6.2 Calculate mean average precision

If we change the confidence threshold, the number of predicted boxes is expected to change in general, and thus the precision and the recall will change accordingly. A metric, called *average precision* (AP) for a given class, is defined as the average of its precisions over recalls. To calculate

AP, we plot the curve of precision versus recall by changing the confidence threshold, and compute the area under the curve (AUC) as AP.

We use an example illustrated in Fig.10.23 to demonstrate the steps for calculating the AP for a given class. Suppose the test dataset is composed of two images (more images in real situations) and there are 12 ground truth boxes (solid boxes) and 13 predicted bounding boxes (dashed boxes, labeled as A, B, …) for a given class. Each predicted bounding box is attached by a confidence score.

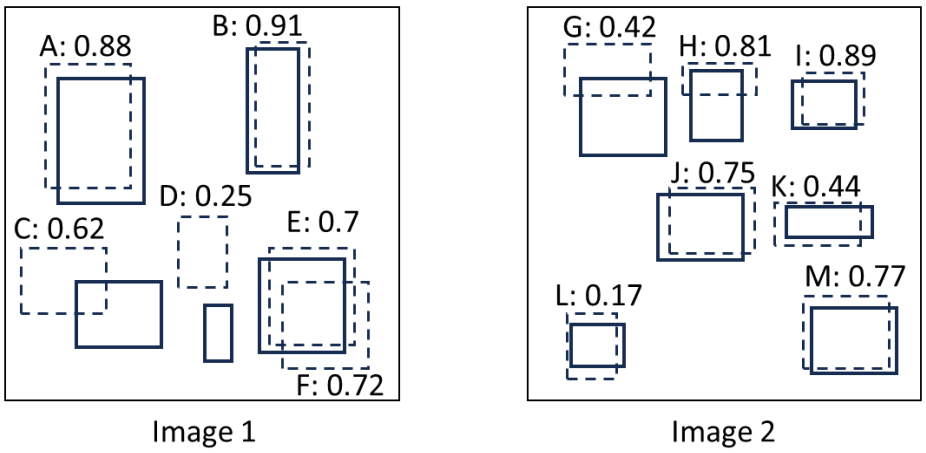


Fig.10.23 ground truth and predictions for a 2-image dataset

A) Plot the precision-recall curve.

To plot the precision-recall curve, we rank all predicted boxes by the confidence score, as shown in Table 10.4. At a particular index, precision is the proportion of all predicted boxes up to the index which are true positive, while recall is the proportion of all ground truth boxes which have been correctly detected up to the index. To calculate precision and recall, we label each box as either TP (true positive) or FP (false positive) by two separate columns and compute the accumulated TP and FP for each index. At each index, the precision is equal to the accumulated TP divided by the index. The recall is equal to the accumulated TP divided by the total number of ground truth boxes (e.g., 12 in this case). The curve of precision-recall is plotted in Fig.10.24.

Table 10.4 Calculate precision and recall

index	image	prediction	confidence	TP	acc TP	FP	acc FP	precision	recall
1	1	B	0.91	1	1	0	0	1	0.083333
2	2	I	0.89	1	2	0	0	1	0.166667
3	1	A	0.88	1	3	0	0	1	0.25
4	2	H	0.81	0	3	1	1	0.75	0.25
5	2	M	0.77	1	4	0	1	0.8	0.333333
6	2	J	0.75	1	5	0	1	0.833333	0.416667
7	1	F	0.72	0	5	1	2	0.714286	0.416667
8	1	E	0.7	1	6	0	2	0.75	0.5
9	1	C	0.62	0	6	1	3	0.666667	0.5
10	2	K	0.44	1	7	0	3	0.7	0.583333
11	2	G	0.42	0	7	1	4	0.636364	0.583333
12	1	D	0.25	0	7	1	5	0.583333	0.583333
13	2	L	0.17	1	8	0	5	0.615385	0.666667

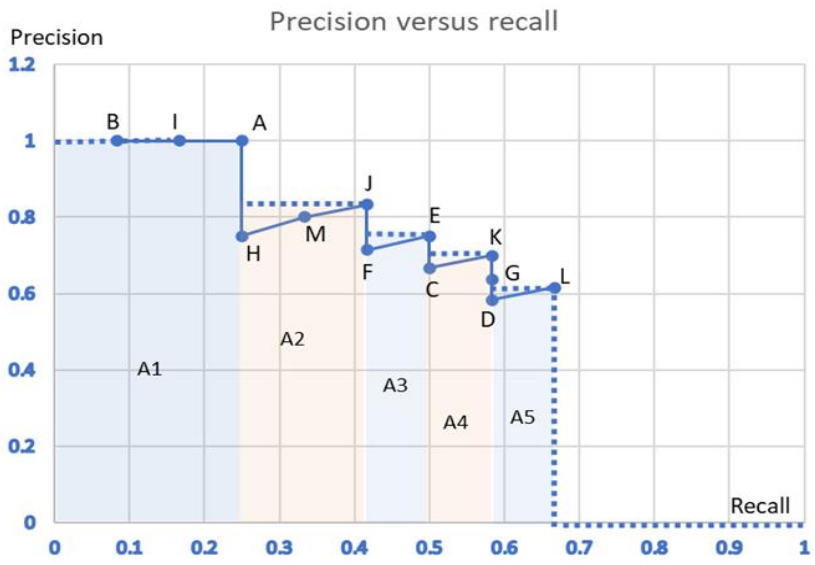


Fig.10.24 Precision-recall curve

B) Interpolate the precision-recall curve.

To simplify the computation of the area under the precision-recall curve, we approximate the curve by the maximum interpolation,

$$P_{interp}(r) = \underset{\tilde{r}:\tilde{r}\geq r}{max}\, P(\tilde{r}) \tag{10.19}$$

where $P(\tilde{r})$ is the precision measured at recall $\tilde{r}$. The interpolated precision at recall r takes the maximum precision measured for which the recall $\tilde{r}$ is greater than r. In other words, we start from the last precision value (at the highest recall) and keep moving toward lower recall, as soon as a higher precision value is found update the precision value with the higher precision. The corresponding interpolated precision in the example is plotted as the dotted line in Fig.10.24.

C) Calculate the average precision.

With the interpolated precision-recall curve, we can calculate the AP by summing up the areas of rectangles under the curve. In Fig.10.24, the AP is calculated as

$AP = A1 + A2 + A3 + A4 + A5 = 1 \times 0.25 + (0.417 - 0.25) \times 0.833 + (0.5 - 0.417) \times 0.75 + (0.583 - 0.5) \times 0.7 + (0.667 - 0.583) \times 0.615 = 0.561$

An alternative way to calculate the AP is to average the sample points of the interpolated precision at 11 recall points: 0, 0.1, 0.2, 0.3, …, 1.0,

$$AP = \frac{1}{11}\sum_{r\in\{0,0.1,\ldots 0.9,1.0\}} P_{interp}(r) \tag{10.20}$$

For the interpolated precision in Fig.10.24, the 11-point AP is calculated as

$$AP = \frac{1}{11}(1 + 1 + 1 + 0.833 + 0.833 + 0.75 + 0.75) = 0.5605$$

In fact, the 11-point average method was introduced in the 2007 PASCAL VOC challenge, and is widely adopted to calculate the AP.

D) Calculate mAP.

The mean average precision (*mAP*) is defined as the average of APs of all classes,

$$mAP = \frac{1}{N}\sum_{k=1}^{N} AP_k \tag{10.21}$$

where N is the number of classes, AP_k is the average precision of class *k*. Note that the value of *mAP* depends on the IOU threshold that was used to determine whether a prediction is true positive or false positive. A major metric for PASCAL VOC challenge is the *mAP* at the IOU threshold 0.5, denoted as $mAP^{@IOU=0.5}$.

Since averaging over IoUs rewards models with better localization, the primary metric for COCO challenge is extended to multiple IOU thresholds, and defined as the average mAP across all 10 IoU thresholds 0.50: 0.05: 0.95 (starting with 0.50, ending at 0.95 with step size 0.05),

$$mAP^{@IOU=[0.5:0.05:0.95]} = \frac{1}{10}\sum_{i=0}^{9} mAP^{@IOU=0.5+0.05\times i} \tag{10.22}$$

As an example, YOLO v3 achieves the following performance on COCO dataset: $mAP^{@IOU=[0.5:0.05:0.95]} = 0.33$, $mAP^{@IOU=0.5} = 0.579$, $mAP^{@IOU=0.75} = 0.344$.

Summary

This chapter describes a popular object detection algorithm – YOLO v1 and its two subsequent versions (v2 and v3). Their architectures have been presented in an evolved order so that we can understand how the performance has been improved through different versions. The key idea of YOLO algorithm is to apply a deep ConvNet (e.g. Darknet) as the backbone network to extract the image feature at a certain grid resolution, and then use a few Conv layers to predict the bounding box and class.

It is essential to understand the loss function of YOLO for training. The total loss consists of three types of losses: confidence loss, box location loss, and probability loss. However, a negative ground truth sample (no object for an anchor box in a grid cell) only results in confidence loss while the prediction for a positive sample generates all three types of losses. The exact definition and implementation of the loss function is very empirical. In general, the box location loss is usually defined as the mean squared error (MSE), and the probability-related loss is defined as entropy loss.

The implementation of YOLO v3 is detailed in Section 10.5. The exploration of the implementation provides a deep understanding of all concepts presented in the preceding sections. In the end, the metric mAP is presented.

Files: C\Users\weido\ch10_object\yolov3_weidong.ipynb

Further Reading

Original papers for Yolo v1 – v3

The authors highly recommend the original papers that provide the origin of YOLO in comprehensive perspectives: motivations, architectures, training tricks, and performance. These papers include YOLO (v1) (Redmon, J., *et al*., 2015), YOLO9000 (v2) (Redmon, J., *et al*., 2016), YOLO(v3) (Redmon, J., *et al*., 2018). The author's website (Redmon, J. website) and GitHub (Redmon, J., GitHub) include rich resources related to his original work, such as papers, source codes, and pre-trained models.

Hands-on Implementations

The blog posts (Kathuria, A. 2017a) and the GitHub (Kathuria, A. 2017b) present the detailed implementation of YOLO v3 using pre-trained model. The website (Geeksforgeeks website) provides an implementation of YOLO v3 from scratch by PyTorch, including loss function and training.

Advance Yolo versions

To explore advanced YOLO architectures up to YOLO v8, one can read a review paper (Terven, J., R., *et al*., 2023) and then narrow down to the original paper of a particular YOLO architecture. The advanced YOLO versions include Yolov4 (Bochkovskiy, A. 2020), Yolov5 (Jocher, G. 2020), YoloR (Wang, C.-Y. 2021), YoloX (Ge, Z. 2021), Yolov6 (Li, C. 2022), Yolov7 (Wang, C.-Y. 2022), Yolov8 (Jocher, G. 2023). AlexeyAB repository (Bochkovskiy, A., GitHub) provides yolov4 source code.

References

Bochkovskiy, A. (GitHub), https://github.com/AlexeyAB/darknet

Bochkovskiy, A., Wang, C.-Y. and Liao, H.-Y. M. (2020), Yolov4: Optimal speed and accuracy of object detection, arXiv:2004.10934 [cs.CV]

Ge, Z., Liu, S., Wang, F., Li, Z. and Sun, J. (2021), Yolox: Exceeding yolo series in 2021, arXiv:2107.08430 [cs.CV]

Geeksforgeeks website, https://www.geeksforgeeks.org/yolov3-from-scratch-using-pytorch/

Jocher, G. (2020), YOLOv5 by Ultralytics, https://github.com/ultralytics/yolov5

Jocher, G., Chaurasia, A., and Qiu, J. (2023), YOLO by Ultralytics, https://github.com/ultralytics/ultralytics

Kathuria, A. (2017a), How to implement a YOLO (v3) object detector from scratch in PyTorch. https://blog.paperspace.com/how-to-implement-a-yolo-object-detector-in-pytorch/

Kathuria, A. (2017b), GitHub, https://github.com/ayooshkathuria/pytorch-yolo-v3

Li, C., Li, L., Jiang, H., Weng, K., Geng, Y., Li, L., Ke, Z., Li, Q., Cheng, M., Nie, W. *et al*. (2022), Yolov6: A single-stage, object detection framework for industrial applications, arXiv:2209.02976 [cs.CV]

Redmon, J., GitHub, https://github.com/pjreddie

Redmon, J., Website, https://pjreddie.com/

Redmon, J., Divvala, S., Girshick, R., and Farhadi, A. (2015) 'You only look once: Unified, real-time object detection', arXiv:1506.02640 [cs.CV]

Redmon, J. and Farhadi, A. (2016) 'Yolo9000: Better, faster, stronger',arXiv:1612.08242 [cs.CV]

Redmon, J. and Farhadi, A. (2018) 'YOLOv3: An Incremental Improvement', arXiv:1804.02767 [cs.CV]

Terven, J., R., and Cordova-Esparza, D., M. (2023), A comprehensive review of YOLO: from YOLOv1 and beyond. https://arxiv.org/pdf/2304.00501.pdf

Wang, C.-Y., Yeh, I.-H. and Liao, H.-Y. M. (2021), You only learn one representation: Unified network for multiple tasks, arXiv:2105.04206 [cs.CV]

Wang, C.-Y, Bochkovskiy, A., and H.-Y. M. Liao, H.-Y.M. (2022), Yolov7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors, arXiv:2207.02696 [cs.CV]

Exercises

10.1 Download and explore the datasets Pascal VOC 2012 and COCO 2017. Write a report about these two datasets.

10.2 Estimate the number of learnable parameters for models yolo v1, v2, and v3. Using the Python program in Section 10.5, verify your estimate of model yolo v3.

10.3 Suppose an image includes two relevant objects: person and dog, with a ground truth label file as [class x y w h]

0 0.3 0.6 0.4 0.8
16 0.5 0.6 0.5 0.3

1) Manually draw the ground truth boxes in a square that represents the image.
2) To compute the loss function, we need to generate a tuple targets = (targets[0], targets[1], targets[3]), where targets[0], targets[1], targets[2] are three tensors for grid scale 13, 26, 52, respectively. Each tensor targets[i] has a shape of ([3, s, s, 6]), as shown in Fig.10.15 for scale s=13. Calculate the value of the targets for this image, i.e., targets[i][anchor_idx, m, n,:], where i=0,1,2, is the grid scale index for scale 13, 26, 52, respectively,
anchor_idx =0,1,2 is the anchor box index at grid scale i,
(m,n) is the grid cell index.

Anchor box assignment

Grid scale index	**Anchor index**		
	0	1	2
0 (for 13x13)	(0.28, 0.22)	(0.38, 0.48)	(0.9, 0.78)
1 (for 26x26)	(0.07, 0.15)	(0.15, 0.11)	(0.14, 0.29)
2 (for 52x52)	(0.02, 0.03)	(0.04, 0.07)	(0.08, 0.06)

10.4 Given two images: 000001.jpg and 000017.jpg, and their label files 000001.txt and 000017.txt. (available at the book website, originally downloaded from https://www.kaggle.com/datasets/aladdinpersson/pascal-voc-dataset-used-in-yolov3-video/?select=PASCAL_VOC).

000001.txt:

16 0.34135977337110485 0.609 0.4164305949008499 0.262
0 0.5070821529745043 0.508 0.9745042492917847 0.972

000017.txt:

0 0.48125 0.3557692307692308 0.19583333333333333 0.3763736263736264
17 0.5114583333333333 0.565934065934066 0.6520833333333333 0.7087912087912088

1) Run the program provided in Section 10.5, and detect the objects in two images.
2) Add Python code to compute the loss of YOLO v3 model on the two images for a) random weight model and b) pre-trained model. The pre-trained model should have a much smaller loss than the random weight model. (hint: the loss is defined in Section 10.4.2. Since you may need to scale the image to fit the model input size, you need to scale the ground truth boxes as well. For Python implementation, please refer to Geeksforgeeks website, https://www.geeksforgeeks.org/yolov3-from-scratch-using-pytorch/).

10.5 Suppose the ground truth bounding boxes for a batch are given as a tensor gtbox with a shape of ([N,6]), where N is the total number of ground truth boxes for the batch, and the format of each ground truth box gtbox[i,:] is

Img_index	x1	y1	x2	y2	Class_index

The predicted bounding boxes are stored in a tensor pbox with a shape of ([M,7]), where M is the total number of predicted boxes for the batch, and the format of each predicted box pbox[j,:] is

Img_index	x1	y1	x2	y2	Class_index	confidence

Write a Python program to compute the mAP for a particular IOU threshold.

10.6 project 1. Train yolo v3 on Pascal VOC dataset from scratch.

10.7 project 2. Train yolo v3 on a custom dataset.