Chapter 12

Object Detection

12.1 Learning Objectives

In the previous chapters, we introduced many models for image classification. In image classification tasks, we assume that there is only one main target in the image and we only focus on how to identify the target category. However, in many situations, there are multiple targets in the image that we are interested in. We not only want to classify them, but also want to obtain their specific positions in the image. In computer vision, we refer to such tasks as object detection. Object detection is widely used in many fields. For example, in self-driving technology, we need to plan routes by identifying the locations of vehicles, pedestrians, roads, and obstacles in the captured video image. Robots often perform this type of task to detect targets of interest. Systems in the security field need to detect abnormal targets, such as intruders or bombs.

Recent years have seen people develop many algorithms for object detection, some of which include YOLO, SSD, Mask RCNN and RetinaNet. In this chapter we will learn YOLO, one of the fast algorithms. 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.

12.2 Object Localization

The task of image classification is to label an image into one of the pre-defined categories. For example, to classify the picture as “car” or “non-car”. However, an algorithm for classification with localization will not only label as picture, but also give the location of the target object (e.g. car) by delivering the location and size of a rectangle around the object. A problem of object detection is to deal with the situation where multiple objects of different categories (e.g. cars, pedestrians, etc.) may present in one picture. The algorithm of the detection is to label all detected objects and their locations.

(a) classification: “car” (b) classification with localization (c) detection

Fig.1 classification, localization and detection

To explain classification with localization, let's consider a picture with a car, shown in Fig.2. A ConvNet can be designed to extract the features of the input image, and a softmax layer can be attached to the ConvNet as the output layer for a classification purpose. For instance, with three classes (1. pedestrian, 2. car, 3. motorcycle), the softmax layer would give four outputs, each of which represents the probability of the corresponding class. For a task of classification with localization, the output layer should deliver the prediction, denoted by $\hat{y}$, on whether an object presents, which object if any and where the object is located. The location of an object is specified by a bounding box which is a rectangle defined by bx, by, bH, bW. The coordinate of the up left corner is (0,0), and the one of the bottom right corner is (1,1). The center of the bounding box is at (bx, by), and the height and width of bounding box are bH and bW, respectively.

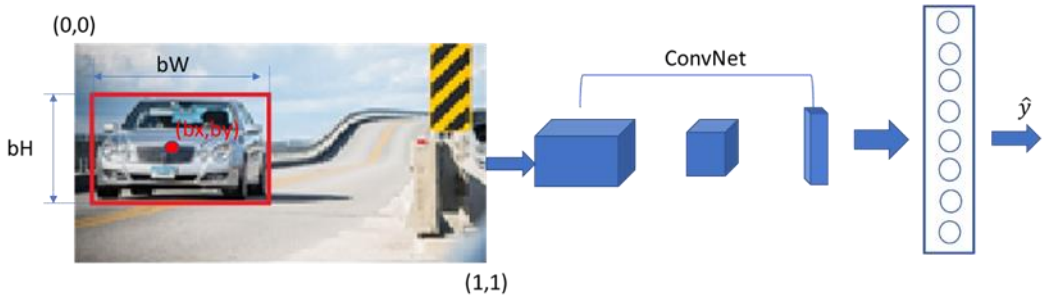


Fig.2 classification with localization

To train the network, the label of each training example, y, has a format

$$y = \begin{bmatrix} pc \\ bx \\ by \\ bH \\ bW \\ c1 \\ c2 \\ c3 \end{bmatrix}$$

pc → If object presents or not (1/0)

bx, by, bH, bW → Bounding box

$c1, c2, c3$ → One-hot code class

For example, the image in Fig.2 could have a label as $y = \begin{bmatrix} 1 \\ 0.2 \\ 0.45 \\ 0.5 \\ 0.4 \\ 0 \\ 1 \\ 0 \end{bmatrix}$. If no target object in the image, pc is set to 0, and other elements in y are set to "don't care". The element pc in the prediction $\hat{y}$ represents the probability of a target object in the input image.

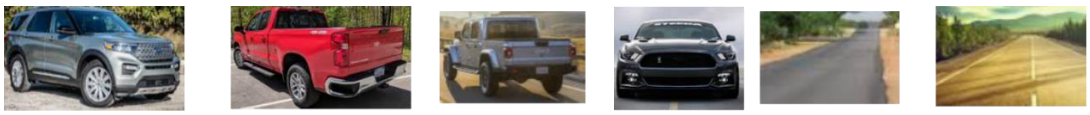
For the sake of simplicity, the loss function can be defined as square error. However, in practice, softmaxt loss may be used for c1, c2 and c3. Square error may be used for bounding box error. Logistic regression (or square error) may be used for pc error.

$$\ell(\hat{y}, y) = \begin{cases} \|\hat{y} - y\|_2 & y_1\ (i.e.\ pc) = 1 \\ (\widehat{y_1} - y_1)^2 & y_1 = 0 \end{cases} \quad (1)$$

12.3 Sliding Windows Detection

12.3.1 The idea of sliding windows detection

An easy intuitive way for object detection is sliding windows detection. The idea of sliding windows detection is to cut the entire image into many smaller pieces, and then classify each piece separately. Let's consider a car detection example. First, we train a ConvNet to classify window-size images as "car" or "no car". To train this ConvNet, we create the positive examples by tightly cropping "car" from regular images including a car, as shown in Fig.3.



Label: y=1 y=1 y=1 y=1 y=0 y=0

Fig. 3 examples for training window ConvNet

Then we apply a window to cut a piece of the image to feed the trained ConvNet to detect whether a car presents in this window. A re-size operation is needed if the window size does not match the input size of the ConvNet. The repetitive process can be performed by sliding the window in the horizontal or vertical direction. After the window have scanned the entire image by sliding, we can change the size of the window to repeat the scanning process. To achieve a reasonable detection resolution (fine regularity), a smaller sliding stride is preferred. Furthermore, different sizes of window are needed to accommodate objects with different sizes. Since each window portion is passed through the ConvNet independently, high computation cost is the major disadvantage of this sliding windows detection method.

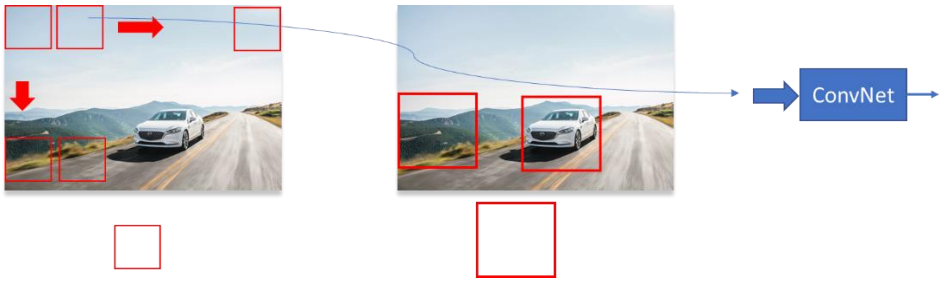


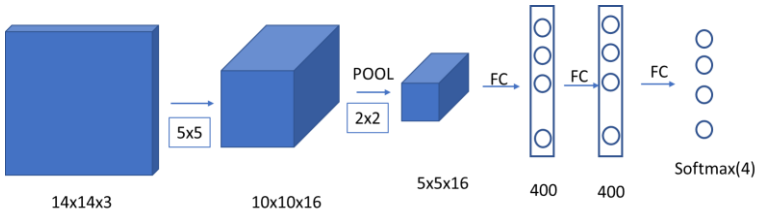
Fig. 4 sliding windows detection

12.3.2 Convolutional implementation of sliding window

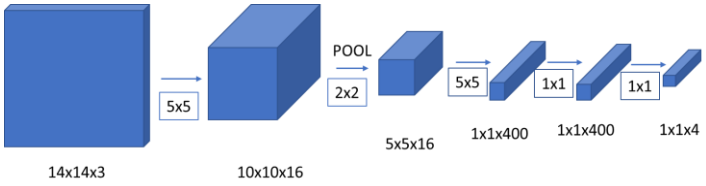
In this section, we will describe how to use convolutional operation to efficiently implement sliding window detection [1]. In contrast to the sliding-window approaches that compute an entire pipeline for each window of the input one at a time, ConvNets are inherently efficient when applied in a sliding fashion because they naturally share computations common to overlapping regions

First, let's identify that the fully connected layers in the classification network can be implemented as 1x1 kernel convolution layers. Fig.5 (a) shows a classification network who's last three layers are fully connected, and (b) shows its equivalent implementation by replacing the fully connected layers with 1x1 kernel convolution operations. The entire ConvNet is then simply a sequence of convolutions, pooling, and activations exclusively.

Then, when applying the trained ConvNet to larger images at test time, we simply apply each convolution over the extent of the full image. This extends the output of each layer to cover the new image size, eventually producing a map of output class predictions, with one spatial location for each "window" (field of view) of input. This is illustrated in Fig.6. The max-pool 2x2 results in an equivalence of sliding window with a stride of 2.



(a)



(b)

Fig. 5 Turning fully connected layers to 1x1convolution layers

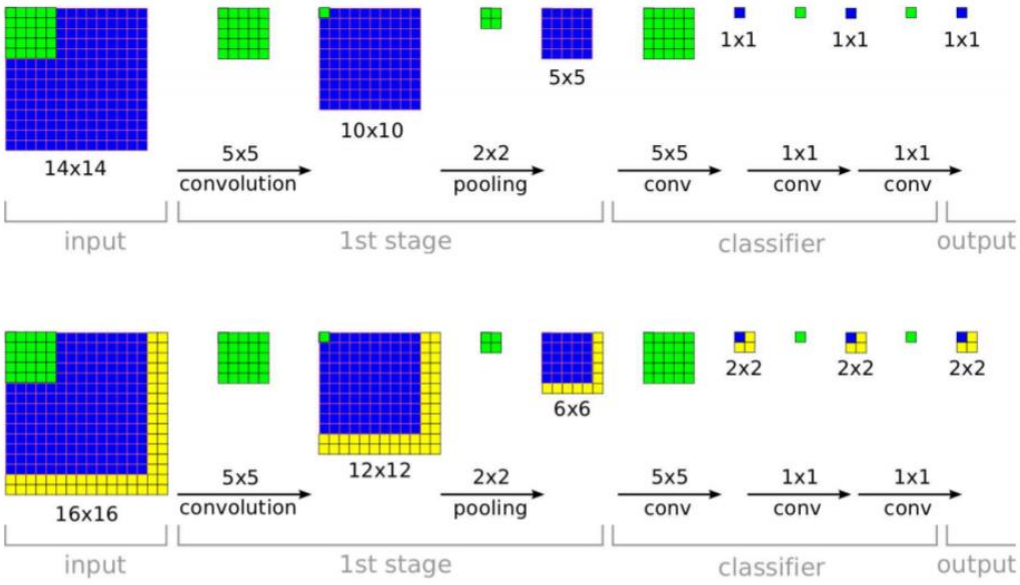


Fig.6: The efficiency of ConvNets for detection (from [1]). During training, a ConvNet produces only a single spatial output (top). But when applied at test time over a larger image, it produces a spatial output map, e.g. 2x2 (bottom). Since all layers are applied convolutionally, the extra computation required for the larger image is limited to the yellow regions. This diagram omits the feature dimension for simplicity.

Now, let's apply the trained filters (or kernels) to a larger image (28x28x3), as shown in Fig.7. Since a input size of 14x14 results in a 1x1x4 tensor output as class prediction (shown in Fig.5(b)), a input image with size 28x28 will give a output tensor 8x8x4. This corresponds to a window (14x14) sliding with a stride 2 across the image, and each vector [i, j, :] in the output tensor represents the detection prediction at its corresponding sliding window position [i, j].

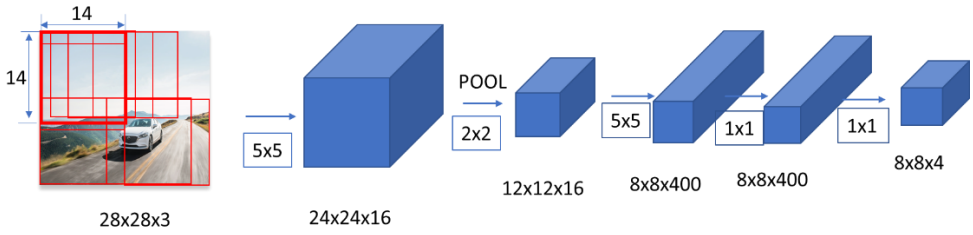


Fig.7 ConvNet implementation of sliding windows detection

12.4 Basic Concepts for YOLO Objection Detection

In general, the bounding box predicted by the sliding window approach usually does not fit well for a detected object because the windows only appear at some fixed sampled locations and their sizes are pre-defined independent of the object's size. Furthermore, an object likely presents in multiple sliding windows, and thus resulting in multiple detections for one object. An efficient and widely used algorithm for predicting a more accurate bounding box is YOLO algorithm. YOLO stands for "You Only Look Once", which was proposed in [2]. With the combination of the ideas of localization and sliding windows detection, we are ready to study the basic YOLO algorithm. In this section, we will present some important concepts for YOLO algorithm.

12.4.1 Basic ConvNet for both object detection and localization

Before we address the ConvNet architecture and training for YOLO algorithm, it is important to understand how we label the training examples.

For simplicity, we assume that the entire image be divided into 3x3 grids (in practice, finer grids are applied, e.g. 19x19 grids), as show in Fig.8. The bounding box of an object is specified by its center (bx, by), height (bH) and width (bW). We label each grid as an eight-element vector y, which indicates whether there is an object (pc=1 for yes), the bounding box parameters and the class of the presented object. At this moment, no more than one object in a grid is assumed. We will remove this assumption in a later section. The bounding box of an object is assigned to a grid at which its center is located. (bx,by) is specified as the relative location within the grid (up left corner is (0,0) and bottom right corner is (1,1)). bH and bW are the relative length of the grid. Note that bH and/or bW are more than one if the bounding box is larger than the grid. If there is no object in a grid, the pc in the corresponding label y is set to 0 and other elements are "don't care". Thus, the total label for the image is a tensor 3x3x8.

Suppose the input images have a size of 100x100x3 and use 3x3 grids to label the images as shown in Fig.8. We can construct a ConvNet to deliver an output tensor of 3x3x8 as shown in Fig.9, and then train the ConvNet using the examples labeled in the way in Fig.8. At test time, the trained ConvNet can detect objects in a regularity (or resolution) of 3x3 grids.

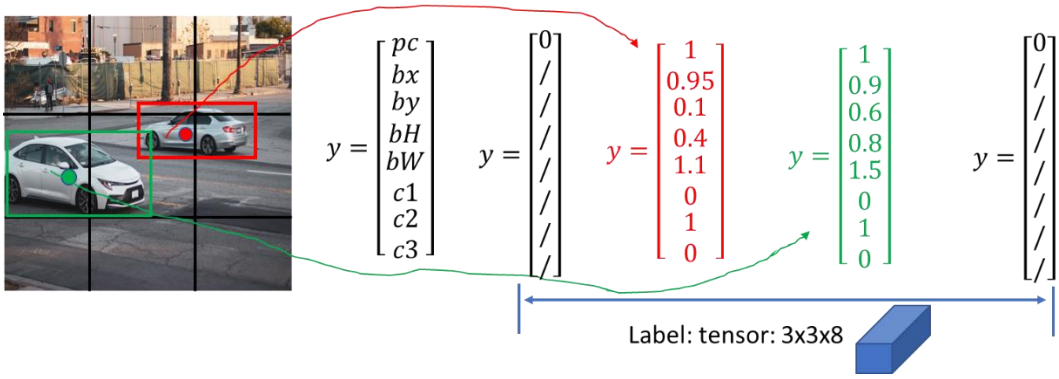


Fig. 8. Training example label for YOLO algorithm

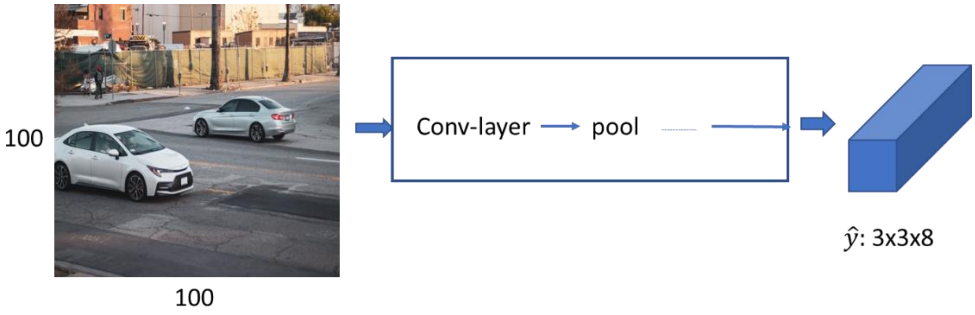


Fig.9 Basic ConvNet for detection and localization

12.4.2 Non-max suppression

So far we have not considered two situations, which are usually the case in practice: 1) an object may be detected by multiple grids when the object is larger or the grid size is relatively small; and 2) multiple objects may present in the same grid when two objects are so close. In this section, we will address the first issue: multiple detection of an object, that is, how to delete the redundant bounding boxes while keeping the most trustable predicted bounding box. For instance, when we increase the grids to 7x7 grids, the prediction tensor will be 7x7x8. It is possible that two or more vectors in the tensor indicate object detections (with a probabilities) and locations (bounding box) due to the same object, as shown in Fig.10.

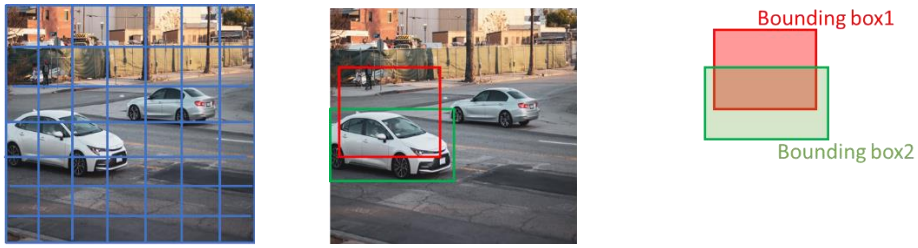
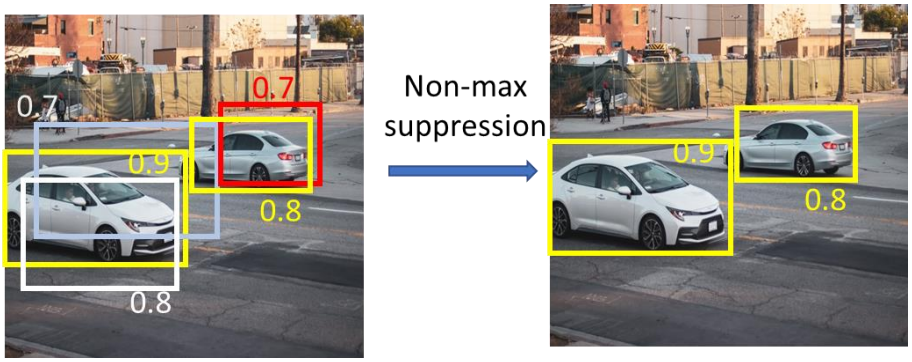


Fig. 10 Intersection over union (IOU)

First let's define a quantity to measure how similar of two bounding boxes are. This quantity is called intersection over union (IOU). Fig.10 shows an example with two predicted bounding boxes (red) for the white car. The IOU is defined as the ratio of their intersection area to their union area. The value of any IOC lies in the range of (0,1]. IOC with a value one implies a perfect match

while a small value close to zero means a small overlap. Thus, IOC 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 an object.



(a) (b)

Fig.11 Non-max suppression

Now we will introduce non-max suppression algorithm to remove the redundant predicted bounding boxes. Consider an example in Fig.11. We assume that only one class (e.g. car) is considered in our discuss for simplicity. In this example, there are obviously two cars detected. One car generates three bounding boxes while the other results in two bounding boxes, as shown in Fig.11(a). Each bounding box is specified by a vector in the predicted tensor 7x7x8. The format of vector is

$$\hat{y} = \begin{bmatrix} pc \\ bx \\ by \\ bH \\ bW \\ c1 \\ c2 \\ c3 \end{bmatrix}$$

probability

Bounding box

One-hot code class

The purpose of non-max suppression algorithm is to keep the maximal probability bounding box while deleting the redundant boxes for each detected object. In the case shown, the resulting bounding boxes are displayed as yellow in Fig.11(b). The non-max suppression algorithm can be described below.

Algorithm: non-max suppression

Step 1: Among the output tensor, discard all bounding boxes with pc less than a threshold (e.g. 0.6)

Step 2: do the followings while there are any remaining bounding boxes:

2.1 Among these remaining boxes, pick the box with the largest pc, and output this box as a prediction, and then remove this box from “remaining bounding boxes”.

2.2 Discard any remaining box with IOU≥ a threshold (e.g. 0.5) with the output box in the step 2.1

If we want to detect multiple objects in three different classes, say pedestrian, car and motorcycle, we just need to apply the non-max suppression independently three times with one on each of the output classes.

12.4.3 Anchor boxes

To detect multiple objects in one grid cell, we introduce multiple, say B, anchor boxes for each grid cell. An *anchor box* is a pre-defined shape of bounding box with width p_w and height p_h. The values of p_w and p_h were determined by running K-means across the entire dataset (we don't need to worry about it at this time). Each anchor box in a grid typically has a different aspect ratio to accommodate different shapes (e.g. fat or skinny) of objects. Note that the sizes of the bounding boxes are generally not equal to the sizes of anchor boxes. Fig.12 shows how to label a training example with grids 3x3 and B=2. Anchor box 1 is applied for upright (or skinny) shape objects while anchor 2 for objects with a shape of flat. The pedestrian and the car accidentally locate at the same grid cell. Thus, the label y for this grid cell is a vector of 2x8 elements, as shown at the right of Fig.12. The first 8 elements are associated anchor 1 for the pedestrian, and the second 8 elements are associated with anchor 2 for the car. The prediction feature map (i.e. output) of the ConvNet is a tensor 3x3x(2x8). Note that, in YOLO, bx and by are the offsets to the top-left corner of the predicting grid cell and normalized by the grid size, and bh and bw are relative and normalized to the entire image.

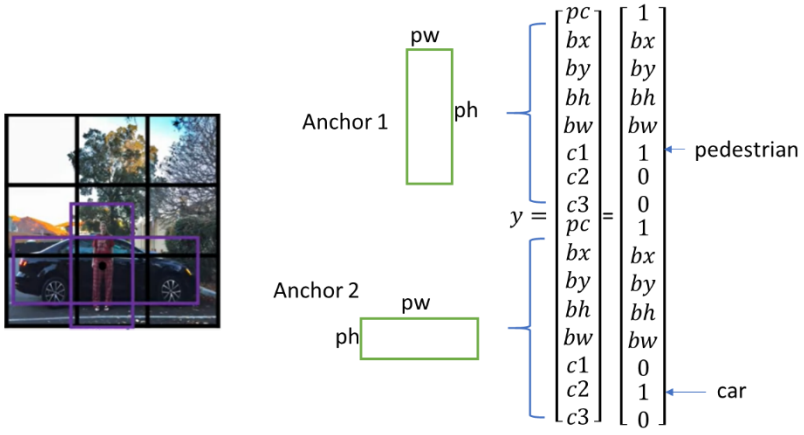


Fig.12 anchor boxes

12.4.4 Output of ConvNet

To implement a specific ConvNet for an object detection task (in Fig.9), it is essential to understand the output of the ConvNet, i.e. the prediction feature map. In this section, we will interpret the prediction feature map and how it can be used to determine the final bounding boxes.

A) Dimension size of prediction feature map

In practice, more grids and more anchor boxes are applied for an acceptable performance. For instance, YOLY v3 uses three anchors and three different grid scales: 13x13, 26x26, and 52x52. Three anchors allow a maximum of three objects to be detected in one grid cell. Three grid scales allow a large range of detectable object sizes. The finer grids, the smaller objects can be detected. For the case specified by grids scale SxS, B anchors and C classes, the prediction feature map is a tensor SxSx(Bx(5+C)). Each bounding box has (5+C) elements. There are B bounding boxes in each grid cell. There are totally SxS grid cells. For example, the prediction feature map is a tensor

13x13x(3x85)=13x13x255 if S=13, B=3, and C=80. Fig.13 illustrates the data structure of prediction feature map.

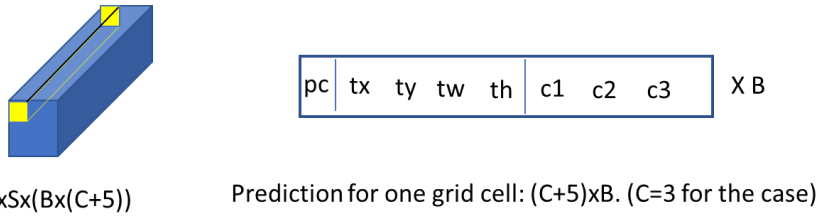


Fig. 13 Prediction feature map. Right: the entire prediction feature map, the yellow volume indicates the prediction for one grid cell. Left: one predicted bounding box for C=3.

B) Confidence scores

In Fig.13, for each bounding box in a grid cell, the object confidence score, *pc*, represents the probability that an object is contained inside the bounding box. Note that a sigmoid function is applied so that it can be interpreted as a probability. Class confidences (e.g., c1, c2, c3, etc.) represent the probabilities of the detected object belonging to a particular class (pedestrian, car, motorcycle etc). Before v3, YOLO used to softmax the class scores with the assumption that the classes are mutually exclusive. However, this assumption may not hold when we have classes like *Women* and *Person*. Thus, YOLO v3 used sigmoid for the class scores.

C) Bounding box

A transform is required to convert the prediction data (tx,ty,tw, th) to a geometrical parameters (i.e. center, width, and height) of the bounding box. It might make sense to directly predict the width and the height of the bounding box, but in practice, that leads to unstable gradients during training. Instead, YOLO v3 predicts the bounding box through a transform applied to the corresponding feature map data. Assume that bx, by, bw, bh are the x,y center co-ordinates, width and height of our prediction. tx, ty, tw, th is what the network outputs associated with the object. cx and cy are the top-left co-ordinates of the grid. pw and ph are anchors dimensions for the box. The bounding box predictions can be obtained by the following transforms, as show in Fig.14.

$$b_x = c_x + \sigma(t_x) \tag{2.a}$$

$$b_y = c_y + \sigma(t_y) \tag{2.b}$$

$$b_w = p_w e^{t_w} \tag{2.c}$$

$$b_h = p_h e^{t_h} \tag{2.d}$$

Thus, note that YOLO doesn't predict the absolute coordinates of the bounding box's center. It predicts offsets which are: 1) relative to the top left corner of the grid cell that is predicting the object, and 2) normalized by the dimensions of the cell from the feature map, which is, 1. For example, if $(\sigma(t_x), \sigma(t_y))$ is (0.7, 0.8), then the center of the predicted bounding box is located at (2.7, 2.8), the red dot shown in Fig.14. In contrast, the resultant predictions, bw and bh, are normalized by the height and width of the image. If the predictions (b_w, b_h) is (0.28, 0.6) for the red bounding box in Fig.14, the actual bounding box width and height are 5x0.28 and 5x0.6 grid cells, respectively.

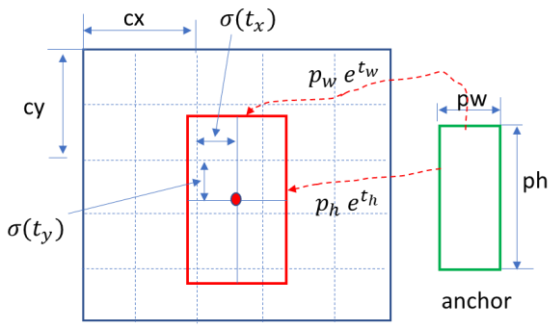


Fig.14 transform ConvNet output feature to bounding box. Input: tx,ty,tw,th,pw,py, cx,cy. The red rectangle is the resulting bounding box.

D) Non-max suppression

An object is usually detected by multiple grid cells with different confidence scores. Non-max suppression we discussed earlier can be applied to remove redundant bounding boxes for each detected object.

12.5 Implementations of YOLO Algorithm

In the previous sections, we have learned the basic concepts/components for YOLO algorithm. In many cases it is impractical in terms of memory and compute power for one to train an Imagenet CNN from scratch. Most deep object detectors consist of a feature extraction CNN (usually pre-trained on Imagenet and fine-tuned for detection) connected to a final layer that reshapes the features into the detector-specific output tensor. It is of practice to use an open-sourced, prebuilt model, adjusting the last layers and the loss functions to accomplish our task.

In this section, we will put all together by presenting the detailed implementations of YOLO algorithm. First, we will introduce PascalVOC dataset to explain the annotation of training images. Secondly, we will describe the architecture and loss function for YOLO v3

12.5.1 Annotation of images: PascalVOC dataset

To train a ConvNet for objection detection, we need to label the training images. We take PascalVOC (visual object classes) dataset [3] as an example to see how an image can be labeled and how to convert this label to the input format for ConvNet. PascalVOC 2007 dataset (train/validation/test) has 9,963 images containing 24,640 annotated objects in 20 classes. PascalVOC 2012 dataset has 11,530 images containing 27,450 annotated objects in 20 classes.

As shown in Fig.15, for each image, PascalVOC provides an annotation file (*.xml) containing the bounding box coordinates of objects in one of 20 classes. PascalVOC encodes bounding boxes by the top-left (xmin, ymin) and bottom-right (xmax, ymax) corner coordinates, but some object detection algorithms encode boxes using the center xy-coordinates with the width and height.

To feed an image into a ConvNet, the image is resized to be square. Correspondingly, we should adjust the PascalVOC bounding box to locate the resized object. We normalize the annotated bounding box coordinates by the image width W and height H, in either corner style or center style as

$$b_{VOC} = \begin{bmatrix} x_{min} \\ y_{min} \\ x_{max} \\ y_{max} \end{bmatrix} \in \mathbb{R}^4 \;\rightarrow\; b_{corner} = \begin{bmatrix} \frac{x_{min}}{W} \\ \frac{y_{min}}{H} \\ \frac{x_{max}}{W} \\ \frac{y_{max}}{H} \end{bmatrix} \in [0,1]^4 \quad or \quad b_{center} = \begin{bmatrix} \frac{x_{min}+x_{max}}{W} \\ \frac{y_{min}+y_{max}}{H} \\ \frac{x_{max}-x_{min}}{W} \\ \frac{y_{max}-y_{min}}{H} \end{bmatrix} \in [0,1]^4 \qquad (3)$$

Since there are 20 classes in PascalVOC, we use an integer $c \in [1,20]$ to represent a class, according to the list: [aeroplane, bicycle, bird, boat, bottle, bus, car, cat, chair, cow, diningtable, dog, horse, motorbike, person, pottedplant, sheep, sofa, train, tvmonitor]. Thus, for each image, the label is a list of objects represented by normalized 4-dimensional bounding box **b** and an integer class ID **c**.

```
<annotation>
       <folder>VOC2012</folder>
       <filename>2008_000089.jpg</filename>
       <source>
           <database>The VOC2008 Database</database>
           <annotation>PASCAL VOC2008</annotation>
           <image>flickr</image>
       </source>
       <size>
           <width>376</width>
           <height>500</height>
           <depth>3</depth>
       </size>
       <segmented>1</segmented>
       <object>
           <name>chair</name>
           <pose>Frontal</pose>
           <truncated>0</truncated>
           <occluded>0</occluded>
           <bndbox>
               <xmin>71</xmin>
               <ymin>18</ymin>
               <xmax>307</xmax>
               <ymax>494</ymax>
           </bndbox>
           <difficult>0</difficult>
       </object>
</annotation>
```

Fig.15 image 2008_000089.jpg and its annotation file 2008_000089.xml (from [4])

12.5.2 YOLO v1 [2]

A) Architecture

YOLO v1 is the original YOLO proposed by Joseph Redmon [] in 2016. YOLO v1 has 24 convolutional layers followed by 2 fully connected layers (FC), as shown in Fig.16. Some convolution layers use 1 × 1 reduction layers alternatively to reduce the depth of the feature maps. Leaky ReLU activation is used for all layers except the final layer. The final layer uses a linear activation function. For the last convolution layer, it outputs a tensor with shape (7, 7, 1024). The tensor is then flattened. Using 2 fully connected layers as a form of linear regression, it outputs 7×7×30 parameters and then reshapes to (7, 7, 30), i.e. 2 boundary box predictions per location, which implies that the image is divided into 7x7 grids.

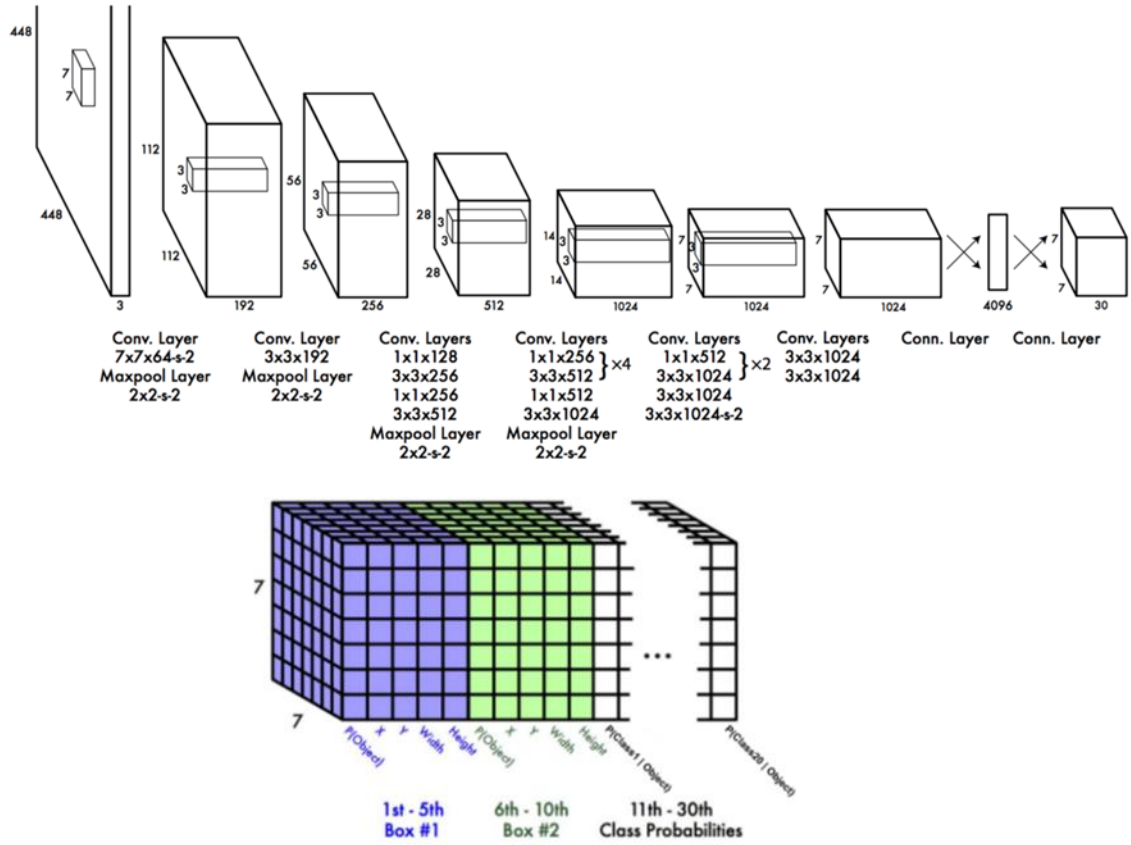


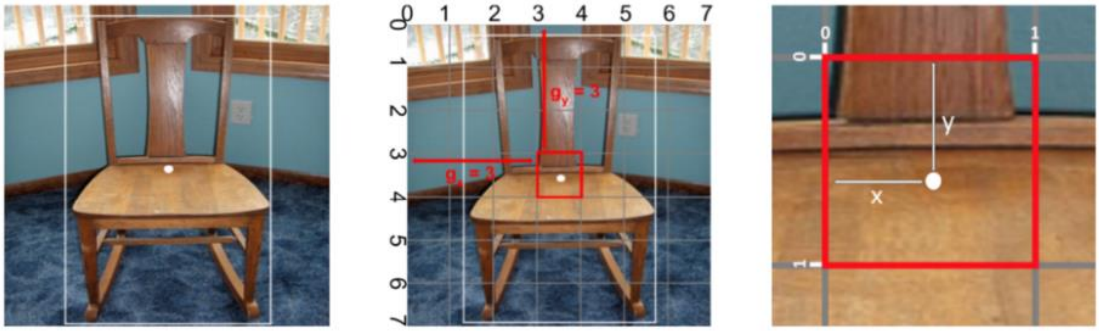
Fig.16 architecture of YOLO and the output feature map

Since the bounding box parameters in the output tensor (7,7,30) are relative to the grid cell and image size, we need to label the training images by converting the center-normalized PascalVOC bounding box encoding to the Yolo bounding box encoding as

$$b_{center} = \begin{bmatrix} x_c \\ y_c \\ w_c \\ h_c \end{bmatrix} \in [0,1]^4 \;\rightarrow\; b_{yolo1} = \begin{bmatrix} f(x_c, g_x) \\ f(y_c, g_y) \\ w_c \\ h_c \end{bmatrix} \in [0,1]^4 , \tag{4}$$

$$where \quad g_x = \lfloor 7 \times x_c \rfloor,\; g_y = \lfloor 7 \times y_c \rfloor,\; f(x_c, g_x) = 7 \times x_c - g_x$$

Thus, a ground truth bounding box can be represented by $b = \begin{bmatrix} x_i \\ y_i \\ w_i \\ h_i \end{bmatrix}, where\ i = 1, 2, \dots, S^2, S = 7\ for\ our\ case$ (i identifies the location of grid cell, xi, yi are offsets to the top-left corner of the grid cell and normalized by the grid size, and wi and hi are normalized by the image size). Fig.17 shows how a ground truth bounding box is assigned for the grid cell (gx=3, gy=3) that is responsible for predicting the object "chair".



(a) given b_{center} (b) assign to cell (g_x, g_y) (c) computer offsets (x,y)

Fig.17 how an object is assigned to a grid cell

B) Loss function

Before we introduce the loss function for training the neural network, let's look at the output feature map of the neural network, a tensor of (7,7,30). Fig. 18 shows the output feature vector for the grid cell (red square) denoted as $\hat{y}$. Each grid cell predicts B (B=2 in our case) bounding boxes ($\widehat{b_1}, \widehat{b_2}$) and confidence scores (c for each) for those boxes. These confidence scores reflect how confident the model is that the box contains an object and also how accurate it thinks the box is that it predicts. Each grid cell also predicts $\hat{P}$ conditional class probabilities $(\widehat{p_1}, \widehat{p_2}, \dots, \widehat{b_{20}})$, Pr(Classi |Object). These probabilities are conditioned on the grid cell containing an object. We only predict one set of class probabilities (i.e. one object) per grid cell, regardless of the number of boxes B.

To define the loss function, we first construct a true feature vector *y* from $\hat{y}$ for each cell having a true object, by following steps, illustrated in Fig.18.

1) We assign the true box b(x,y,w,h) to box1 or box2 based on which predicted bounding box has the highest Intersection over Union with b. The resulting box confidence score c is calculated by

$$c = \max_{\hat{b} \in \{\widehat{b1}, \widehat{b2}\}} IOU(b, \hat{b}) \tag{5}$$

2) We use the index of class to construct the true class probability vector $P \in [0,1]^{20}$, in which all elements are zero except at index, so p[class=9]= 1.

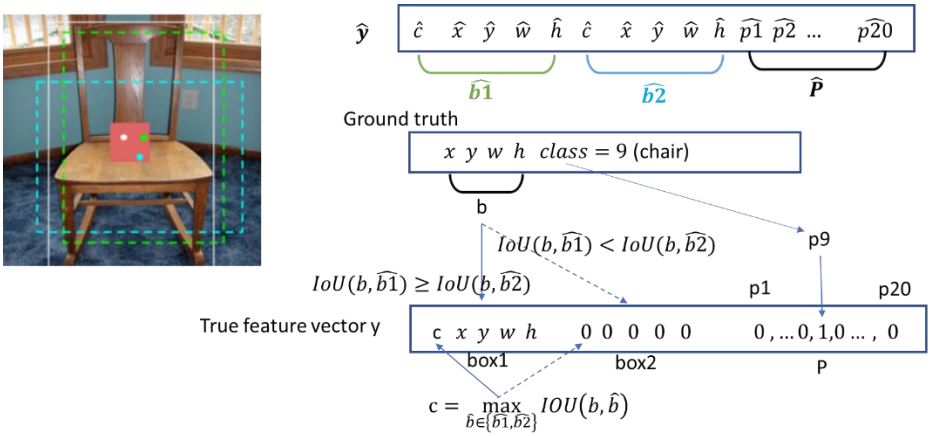


Fig.18 predicted feature vector $\hat{y}$ and true vector y for a cell containing an object

Sum-squared error is the backbone of YOLO's loss design. Since multiple grid cells do not contain any objects and their confidence score is zero. They overpower the gradients from a few cells that contain the objects. To avoid such overpowering leading to training divergence and model instability, YOLO increases the weight (λcoord = 5) for predictions from bounding boxes containing objects and reduces the weight (λnoobj = 0.5) for predictions from bounding boxes that do not contain any objects.

To represent the loss function, we introduce a subscript i for the elements in the true feature vector y, to indicate the grid cell, and i is also applied to corresponding predicting box in $\hat{y}$. The loss function is defined as

$$
\begin{aligned}
&\lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] \\
&+ \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right] \\
&+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(C_i - \hat{C}_i \right)^2 \\
&+ \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i \right)^2 \\
&+ \sum_{i=0}^{S^2} \mathbb{1}_{i}^{\text{obj}} \sum_{c \in \text{classes}} \left(p_i(c) - \hat{p}_i(c) \right)^2
\end{aligned}
\tag{6}
$$

YOLO uses sum-squared error between the predictions $\hat{y}$ and the ground truth y to calculate the loss with the condition whether an object is detected in the box. To understand the loss function, we can divide the loss function into three parts:

1) Localization loss. The localization loss measures the errors in the predicted boundary box locations and sizes. We only count the box responsible for detecting the object.

$$\lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right]$$

$$+ \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right]$$

where

$\mathbb{1}_{ij}^{obj} = 1$ if the j th boundary box in cell i is responsible for detecting the object, otherwise 0.

λ_{coord} increase the weight for the loss in the boundary box coordinates.

Sum-squared error also equally weights errors in large boxes and small boxes. Our error metric should reflect that small deviations in large boxes matter 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.

2) Confidence loss. If an object is detected in the box, the confidence loss (measuring the objectness of the box) is:

$$\sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(C_i - \hat{C}_i \right)^2$$

where

$\hat{C}_i$ is the box confidence score of the box j in cell i.

$\mathbb{1}_{ij}^{obj} = 1$ if the j th boundary box in cell i is responsible for detecting the object, otherwise 0.

If an object is not detected in the box, the confidence loss is:

$$\lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i \right)^2$$

where

$\mathbb{1}_{ij}^{noobj}$ is the complement of $\mathbb{1}_{ij}^{obj}$.

$\hat{C}_i$ is the box confidence score of the box j in cell i.

λ_{noobj} weights down the loss when detecting background.

Ci=0

Most boxes do not contain any objects. This causes a class imbalance problem, i.e. we train the model to detect background more frequently than detecting objects. To remedy this, we weight this loss down by a factor λnoobj (default: 0.5).

3) Classification loss. If an object is detected, the classification loss at each cell is the squared error of the class conditional probabilities for each class:

$$\sum_{i=0}^{S^2} \mathbb{1}_i^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2$$

where

$\mathbb{1}_i^{obj} = 1$ if an object appears in cell i, otherwise 0.

$\hat{p}_i(c)$ denotes the conditional class probability for class c in cell i.

The loss function only penalizes class probabilities error, if an object is present in that grid cell.

C) Inference

Just like in training, predicting detections for a test image only requires one network evaluation. The network predicts 98 (i.e., 7x7x2) bounding boxes per image and 20 class probabilities for each box. YOLO v1 is extremely fast at test time since it only requires a single network evaluation, unlike classifier-based methods.

The grid design enforces spatial diversity in the bounding box predictions. Often it is clear which grid cell an object falls in to and the network only predicts one box for each object. Specifically, each grid cell predicts two bounding boxes with their respective object existence probabilities $\hat{c}$=P(Object) and a class probability distribution $\hat{P}$, so each cell only predicts one object and, at prediction time, we select the bounding box with the highest value of P(Object), which is the probability the box contains an object.

However, some large objects or objects near the border of multiple cells can be well localized by multiple cells. Non-maximal suppression can be used to fix these multiple detections.

D) Limitations

YOLO v1 imposes strong spatial constraints on bounding box predictions since each grid cell only predicts two boxes and can only have one class. This spatial constraint limits the number of nearby objects that the model can predict. The model struggles with small objects that appear in groups, such as flocks of birds.

Since the model learns to predict bounding boxes from data, it struggles to generalize to objects in new or unusual aspect ratios or configurations. The model also uses relatively coarse features for predicting bounding boxes since the architecture has multiple downsampling layers from the input image.

Finally, while we train on a loss function that approximates detection performance, the loss function treats errors the same in small bounding boxes versus large bounding boxes. A small error in a large box is generally benign but a small error in a small box has a much greater effect on IOU. The main source of error is incorrect localizations.

12.5.3 YOLO v2 [5]

YOLO v2 is the second version of the YOLO with the objective of improving the accuracy significantly while making it faster. The improvements in YOLO v2 include: 1) multiple-object prediction per grid cell; 2) the introduce of anchor boxes for localization; 3) upgraded ConvNet and batch normalization.

A) Output tensor

The prediction feature map in YOLO v2 is a tensor of SxSx(Bx(C+5)). For each grid cell, there are B bounding boxes. ConvNet predicts confidence score (pc), location (x,y,w,h) and conditional probabilities (p1,…,p20) of all classes for each box in each grid cell. Thus, the probability that an object of the i-th class is given by pc×pi. If this value is greater than a threshold, we think that the ConvNet predicted that an object of the i-th class exists in this bounding box. Fig. 19 shows the case where S=7, B=5 and C=20. Note that the actual size of the tensor in practice is larger, such as S=13, 19, 26, or 52, C=80.

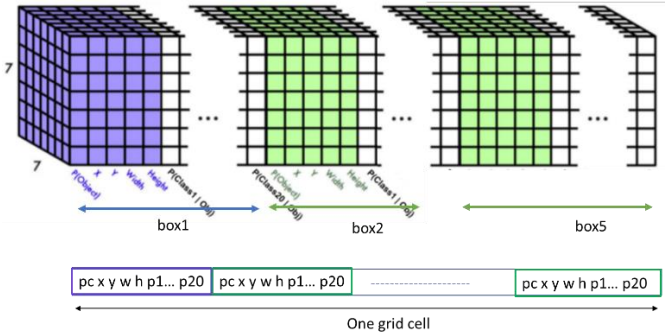


Fig.19 ConvNet output tensor: prediction feature map

B) Anchor Boxes

In YOLO v1, the prediction of the box shape was done arbitrarly without a predefined aspect ratio. That is, the ConvNet did not know what shapes of bounding box are most likely to detect an object of a certain class. For example, when the ConvNet tries to detect humans, it searches humans with square bounding box and vertical rectangle equally likely. However, in the real world, humans usually fit more in vertical rectangles than square boxes. So the ConvNet should search humans with vertical rectangle. With appropriate predefined box shapes (i.e. anchor boxes), we can make it easier for the ConvNet to learn to predict good detections.

Instead of choosing anchor boxes by hand, we run k-means clustering on the training set bounding boxes to automatically find good anchor boxes. In fact, by runing k-means cluster algorithm on the dataset Pascal VOC, we can partition the bounding boxes of ordinary objects into k clusters (e.g. k=5). The mean of each cluster is defined as an anchor box.

For k=5, the five anchor boxes are illustrated in Fig. 20. We associate the 5 prediction bounding boxes with the 5 anchor boxes, respectively. Thus, the prediction bounding boxes predict the width and height offsets relative to the anchor box, instead of direct width and height.

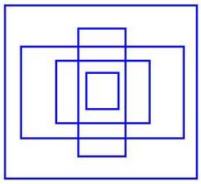


Fig.20 five anchor boxes

The ConvNet predicts 5 bounding boxes at each cell in the output feature map. In addition to the conditional class probabilities, the ConvNet predicts 5 coordinates for each bounding box, tx, ty, tw, th, and to. The relationship between the prediction values and the actual bounding box location is given by equation (2), re-stated here as

$$b_x = \sigma(t_x) + c_x$$
$$b_y = \sigma(t_y) + c_y$$
$$b_w = p_w e^{t_w}$$
$$b_h = p_h e^{t_h}$$
$$Pr(\text{object}) * IOU(b, \text{object}) = \sigma(t_o)$$

where

t_x, t_y, t_w, t_h are predictions made by YOLO.

c_x, c_y is the top left corner of the grid cell of the anchor.

p_w, p_h are the width and height of the anchor.

c_x, c_y, p_w, p_h are normalized by the image width and height.

b_x, b_y, b_w, b_h are the predicted boundary box.

$\sigma(t_o)$ is the box confidence score.

To define the loss function, we label the true feature vectors y for training images as Fig.21. The loss function can be defined in the way as YOLO v1.

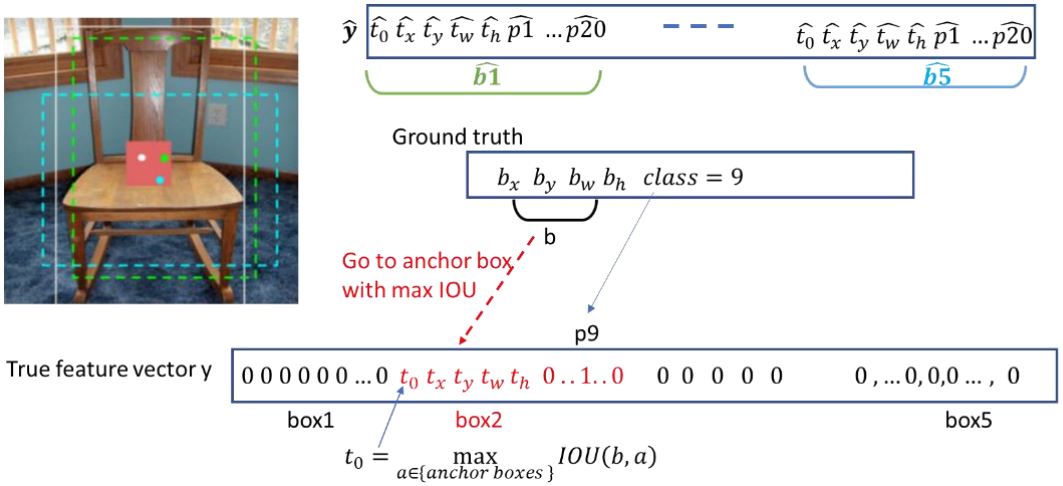


Fig.21 True feature vector y and prediction feature vector $\hat{y}$

C) Inference

A typical YOLO v2 network predicts 13x13x5 bounding boxes per image and 20 class probabilities for each box. Each grid cell can detect up to 5 different objects. The probability of an object of class *i* appearing in a bounding box is equal to $\sigma(\hat{t_0}) \times \hat{p}_i$. If the probability is more than a threshold, the object is detected, and the corresponding bounding box is obtained. Finally, to

remove redundant multiple detections of an object, non-max suppression algorithm is applied to all detected bounding boxes for each class independently.

D) YOLO2 architecture

A widely used architecture for YOLO v2 is the one proposed by the original paper [5] for a good trade-off between model complexity and performance. The YOLO framework uses a custom network as the base feature extractor. This network is called Darknet-19 because it has 19 convolutional layers, as shown in Fig.22. Batch normalization is applied to all convolutional layers to stabilize training, speed up convergence, and regularize the model. The activation for convolutional layers is leakyReLu.

Darknet was trained on the standard ImageNet 1000 class classification dataset for 160 epochs using stochastic gradient descent with a starting learning rate of 0.1, polynomial rate decay with a power of 4, weight decay of 0.0005 and momentum of 0.9. During training, standard data augmentation tricks are used including random crops, rotations, and hue, saturation, and exposure shifts. After the initial training on images at 224 × 224, it was fine-tuned at a larger size, 448. For this fine tuning the model with the above parameters was trained for only 10 epochs and starting at a learning rate of 10^{-3}.

Type	Filters	Size/Stride	Output
Convolutional	32	3 × 3	224 × 224
Maxpool		2 × 2/2	112 × 112
Convolutional	64	3 × 3	112 × 112
Maxpool		2 × 2/2	56 × 56
Convolutional	128	3 × 3	56 × 56
Convolutional	64	1 × 1	56 × 56
Convolutional	128	3 × 3	56 × 56
Maxpool		2 × 2/2	28 × 28
Convolutional	256	3 × 3	28 × 28
Convolutional	128	1 × 1	28 × 28
Convolutional	256	3 × 3	28 × 28
Maxpool		2 × 2/2	14 × 14
Convolutional	512	3 × 3	14 × 14
Convolutional	256	1 × 1	14 × 14
Convolutional	512	3 × 3	14 × 14
Convolutional	256	1 × 1	14 × 14
Convolutional	512	3 × 3	14 × 14
Maxpool		2 × 2/2	7 × 7
Convolutional	1024	3 × 3	7 × 7
Convolutional	512	1 × 1	7 × 7
Convolutional	1024	3 × 3	7 × 7
Convolutional	512	1 × 1	7 × 7
Convolutional	1024	3 × 3	7 × 7
Convolutional	1000	1 × 1	7 × 7
Avgpool		Global	1000
Softmax			

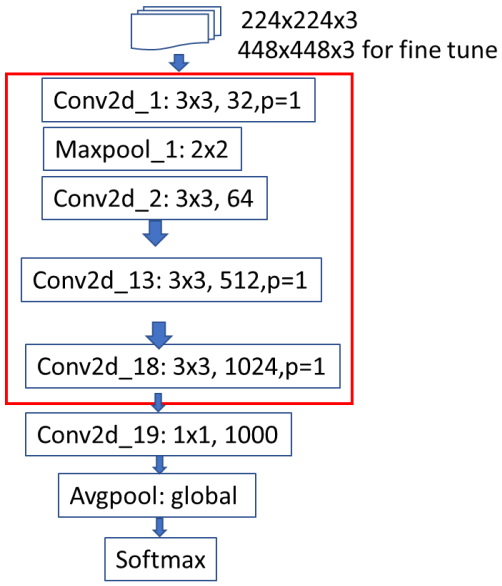


Fig. 22 Darknet-19

YOLO v2 network is obtained by modifying Darknet-19 for detection through removing the last three layers of Darknet-19 (i.e. keeping the layers in the red box in Fig.22) and instead adding on three 3 × 3 convolutional layers with 1024 filters each followed by a final 1 × 1 convolutional layer with the number of outputs we need for detection. For VOC we predict 5 boxes with 5

coordinates each and 80 classes per box so 425 filters. We also add a passthrough layer from the final 3 × 3 × 512 layer to the second to last convolutional layer so that our model can use fine grain features. The resulting YOLO v2 network is shown in Fig.23. This network is re-trained using localization labels with a typical image size 416x416x3. Note that zero bias (bias=False), batch normalization and leakyReLu are applied to all convolutional layers except the last one (Conv2d_23). In Conv2d_23, a bias is applied to filters (bias is set to True). The "reshape" layer after the Conv2d_13 is a reorganization layer. If the input image has dimension 416x416x3, then Conv2d_13 would have an output size of 26x26x512 (HWC). The reorganization layer takes every alternate pixel and puts that into a different channel. Hence, the output of "reshape" layer will be 13x13x256. This output will be concatenated with the output of Conv2d_20 to the feed the layer Conv2d_22.

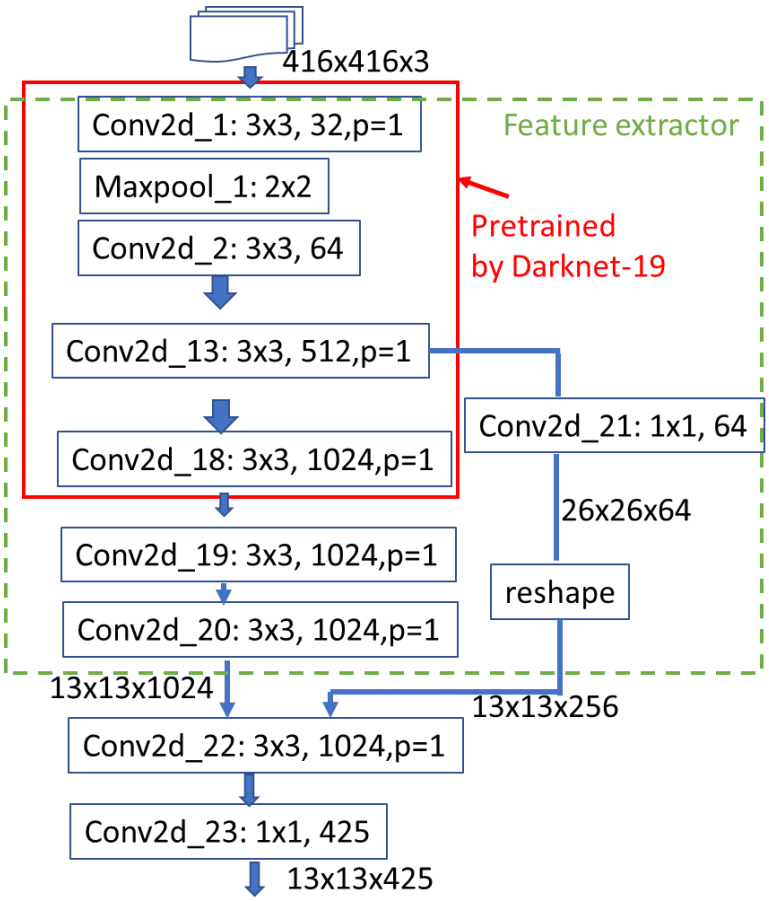


Fig.23 YOLO v2 network architecture: image 416x416, S=13, B=8, C=80. If image size is 608x608, then S=19. If image size is 224x224, then S=7.

The following table summarizes a pre-trained YOLO v2 model for image size 608x608 with a 19x19 grid resolution. For each layer, the name, output shape, number of parameters, and the previous layer are listed. Readers are encouraged to add one column to specify the size of filters for each layer, and check the consistency between the filter size and the number of parameters. Fig.24 shows detected image examples by the model.

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	(None, 608, 608, 3)	0	
conv2d_1 (Conv2D)	(None, 608, 608, 32)	864	input_1
batch_normalization_1 (BatchNor	(None, 608, 608, 32)	128	conv2d_1
leaky_re_lu_1 (LeakyReLU)	(None, 608, 608, 32)	0	batch_normalization_1
max_pooling2d_1 (MaxPooling2D)	(None, 304, 304, 32)	0	leaky_re_lu_1
conv2d_2 (Conv2D)	(None, 304, 304, 64)	18432	max_pooling2d_1
batch_normalization_2 (BatchNor	(None, 304, 304, 64)	256	conv2d_2
leaky_re_lu_2 (LeakyReLU)	(None, 304, 304, 64)	0	batch_normalization_2
max_pooling2d_2 (MaxPooling2D)	(None, 152, 152, 64)	0	leaky_re_lu_2
conv2d_3 (Conv2D)	(None, 152, 152, 128)	73728	max_pooling2d_2
batch_normalization_3 (BatchNor	(None, 152, 152, 128)	512	conv2d_3
leaky_re_lu_3 (LeakyReLU)	(None, 152, 152, 128)	0	batch_normalization_3
conv2d_4 (Conv2D)	(None, 152, 152, 64)	8192	leaky_re_lu_3
batch_normalization_4 (BatchNor	(None, 152, 152, 64)	256	conv2d_4
leaky_re_lu_4 (LeakyReLU)	(None, 152, 152, 64)	0	batch_normalization_4
conv2d_5 (Conv2D)	(None, 152, 152, 128)	73728	leaky_re_lu_4
batch_normalization_5 (BatchNor	(None, 152, 152, 128)	512	conv2d_5
leaky_re_lu_5 (LeakyReLU)	(None, 152, 152, 128)	0	batch_normalization_5
max_pooling2d_3 (MaxPooling2D)	(None, 76, 76, 128)	0	leaky_re_lu_5
conv2d_6 (Conv2D)	(None, 76, 76, 256)	294912	max_pooling2d_3
batch_normalization_6 (BatchNor	(None, 76, 76, 256)	1024	conv2d_6
leaky_re_lu_6 (LeakyReLU)	(None, 76, 76, 256)	0	batch_normalization_6
conv2d_7 (Conv2D)	(None, 76, 76, 128)	32768	leaky_re_lu_6
batch_normalization_7 (BatchNor	(None, 76, 76, 128)	512	conv2d_7
leaky_re_lu_7 (LeakyReLU)	(None, 76, 76, 128)	0	batch_normalization_7
conv2d_8 (Conv2D)	(None, 76, 76, 256)	294912	leaky_re_lu_7
batch_normalization_8 (BatchNor	(None, 76, 76, 256)	1024	conv2d_8
leaky_re_lu_8 (LeakyReLU)	(None, 76, 76, 256)	0	batch_normalization_8
max_pooling2d_4 (MaxPooling2D)	(None, 38, 38, 256)	0	leaky_re_lu_8[0][0]
conv2d_9 (Conv2D)	(None, 38, 38, 512)	1179648	max_pooling2d_4
batch_normalization_9 (BatchNor	(None, 38, 38, 512)	2048	conv2d_9
leaky_re_lu_9 (LeakyReLU)	(None, 38, 38, 512)	0	batch_normalization_9
conv2d_10 (Conv2D)	(None, 38, 38, 256)	131072	leaky_re_lu_9
batch_normalization_10 (BatchNo	(None, 38, 38, 256)	1024	conv2d_10
leaky_re_lu_10 (LeakyReLU)	(None, 38, 38, 256)	0	batch_normalization_10
conv2d_11 (Conv2D)	(None, 38, 38, 512)	1179648	leaky_re_lu_10
batch_normalization_11 (BatchNo	(None, 38, 38, 512)	2048	conv2d_11
leaky_re_lu_11 (LeakyReLU)	(None, 38, 38, 512)	0	batch_normalization_11
conv2d_12 (Conv2D)	(None, 38, 38, 256)	131072	leaky_re_lu_11

batch_normalization_12 (BatchNo (None, 38, 38, 256)	1024	conv2d_12
leaky_re_lu_12 (LeakyReLU) (None, 38, 38, 256)	0	batch_normalization_12
conv2d_13 (Conv2D) (None, 38, 38, 512)	1179648	leaky_re_lu_12
batch_normalization_13 (BatchNo (None, 38, 38, 512)	2048	conv2d_13
leaky_re_lu_13 (LeakyReLU) (None, 38, 38, 512)	0	batch_normalization_13
max_pooling2d_5 (MaxPooling2D) (None, 19, 19, 512)	0	leaky_re_lu_13
conv2d_14 (Conv2D) (None, 19, 19, 1024)	4718592	max_pooling2d_5
batch_normalization_14 (BatchNo (None, 19, 19, 1024)	4096	conv2d_14
leaky_re_lu_14 (LeakyReLU) (None, 19, 19, 1024)	0	batch_normalization_14
conv2d_15 (Conv2D) (None, 19, 19, 512)	524288	leaky_re_lu_14
batch_normalization_15 (BatchNo (None, 19, 19, 512)	2048	conv2d_15
leaky_re_lu_15 (LeakyReLU) (None, 19, 19, 512)	0	batch_normalization_15
conv2d_16 (Conv2D) (None, 19, 19, 1024)	4718592	leaky_re_lu_15
batch_normalization_16 (BatchNo (None, 19, 19, 1024)	4096	conv2d_16
leaky_re_lu_16 (LeakyReLU) (None, 19, 19, 1024)	0	batch_normalization_16
conv2d_17 (Conv2D) (None, 19, 19, 512)	524288	leaky_re_lu_16
batch_normalization_17 (BatchNo (None, 19, 19, 512)	2048	conv2d_17
leaky_re_lu_17 (LeakyReLU) (None, 19, 19, 512)	0	batch_normalization_17
conv2d_18 (Conv2D) (None, 19, 19, 1024)	4718592	leaky_re_lu_17
batch_normalization_18 (BatchNo (None, 19, 19, 1024)	4096	conv2d_18
leaky_re_lu_18 (LeakyReLU) (None, 19, 19, 1024)	0	batch_normalization_18
conv2d_19 (Conv2D) (None, 19, 19, 1024)	9437184	leaky_re_lu_18
batch_normalization_19 (BatchNo (None, 19, 19, 1024)	4096	conv2d_19
leaky_re_lu_19 (LeakyReLU) (None, 19, 19, 1024)	0	batch_normalization_19
conv2d_20 (Conv2D) (None, 19, 19, 1024)	9437184	leaky_re_lu_19
batch_normalization_20 (BatchNo (None, 19, 19, 1024)	4096	conv2d_20
leaky_re_lu_20 (LeakyReLU) (None, 19, 19, 1024)	0	batch_normalization_20
conv2d_21 (Conv2D) (None, 38, 38, 64)	32768	leaky_re_lu_13
batch_normalization_21 (BatchNo (None, 38, 38, 64)	256	conv2d_21
leaky_re_lu_21 (LeakyReLU) (None, 38, 38, 64)	0	batch_normalization_21
space_to_depth_x2 (Lambda) (None, 19, 19, 256)	0	leaky_re_lu_21
concatenate_1 (Concatenate) (None, 19, 19, 1280)	0	(space_to_depth_x2, leaky_re_lu_20)
conv2d_22 (Conv2D) (None, 19, 19, 1024)	11796480	concatenate_1
batch_normalization_22 (BatchNo (None, 19, 19, 1024)	4096	conv2d_22
leaky_re_lu_22 (LeakyReLU) (None, 19, 19, 1024)	0	batch_normalization_22
conv2d_23 (Conv2D) (None, 19, 19, 425)	435625	leaky_re_lu_22

Total params: 50,983,561
Trainable params: 50,962,889
Non-trainable params: 20,672

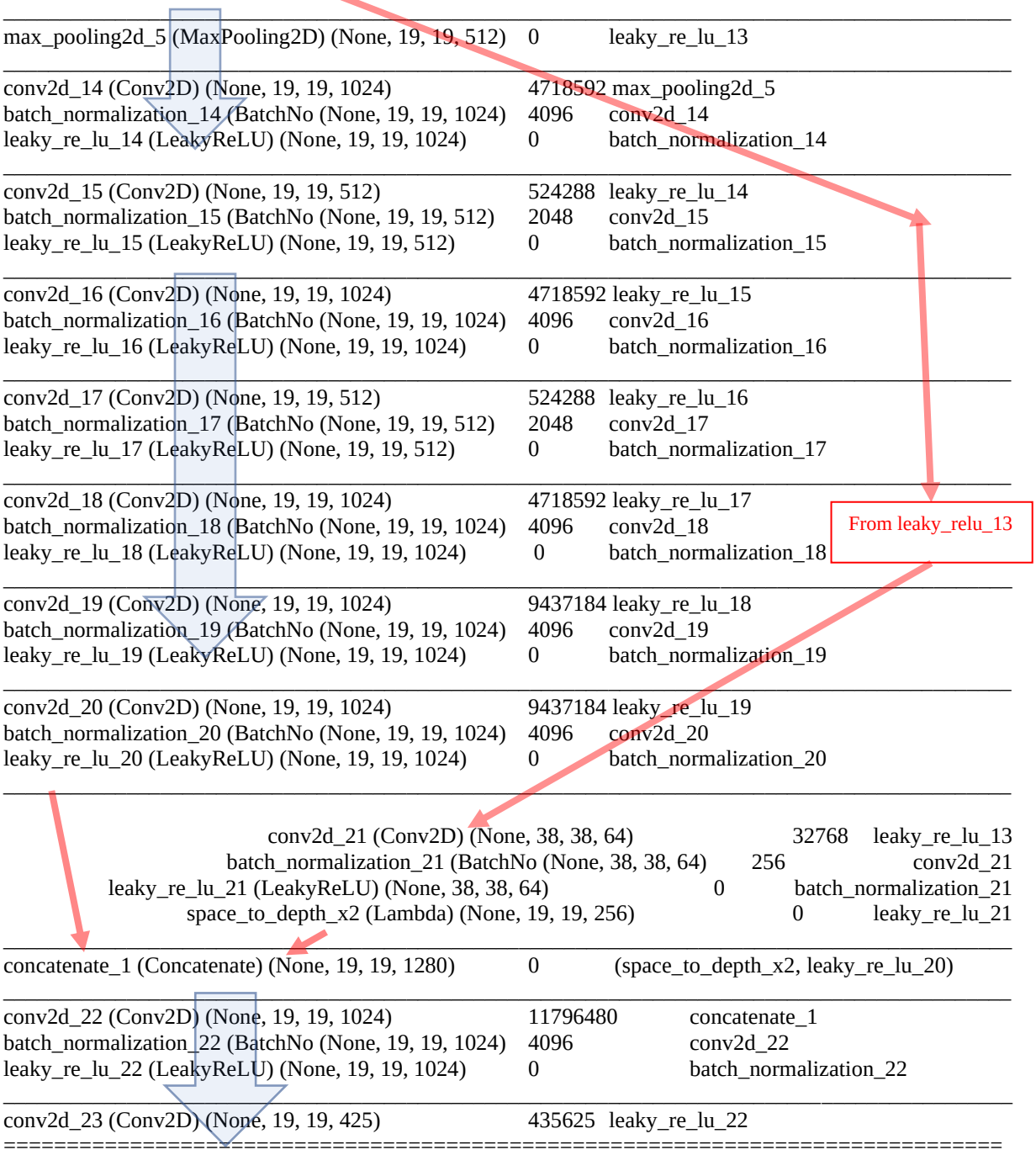


Fig.23 Object detection by YOLO v2

12.5.4 YOLO v3 [6]

A) Architecture of YOLO v3

YOLO v3 is an improvement over previous YOLO detection networks. Compared to prior versions, it features multi-scale detection, stronger feature extractor network, and some changes in the loss function. YOLOv3 uses the multi-scale prediction method to improve the defects of YOLOv2 for small target recognition, significantly improving the recognition accuracy of small targets while maintaining the rapid detection speed of YOLOv2. A typical setting for YOLO v3 is: three scale predictions (13x13, 26x26, 52x52 for image 416x416), three anchor boxes for each scale (B=3), and the number of classes 80 (C=80). The architecture of YOLO v3 is shown in Fig.24.

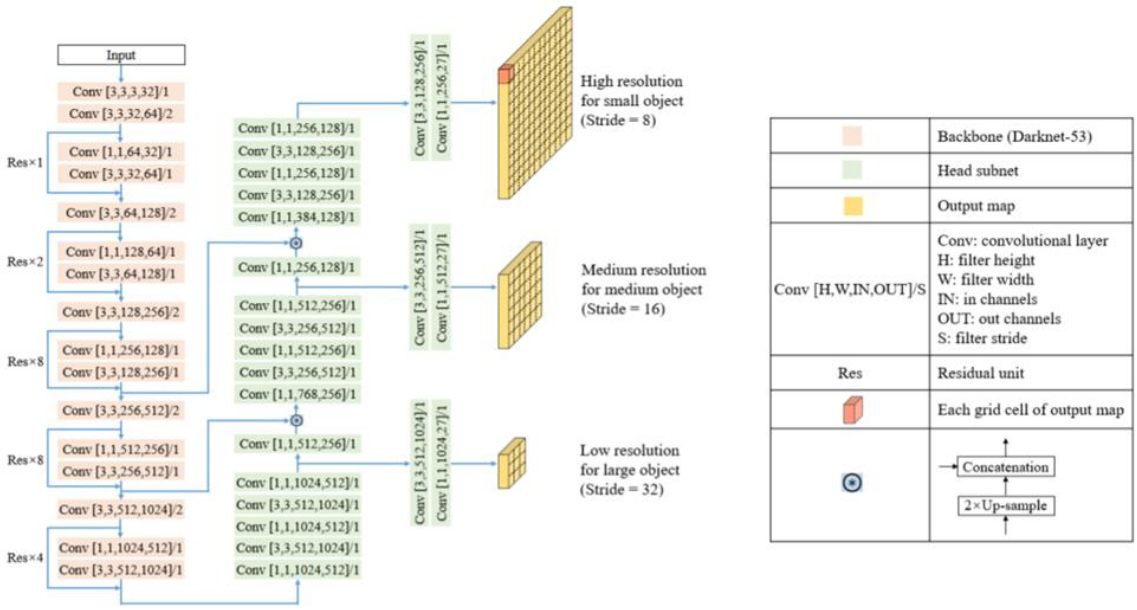


Fig. 24 Architecture of YOLO v3 (from [7])
(the last conv layers should be [1,1,1024,255], [1,1,512,255], [1,1,256,255])

YOLOv3 uses the Darknet-53, instead of Darknet-19, as the backbone for a more accurate feature extraction. Like Darknet-19, the last three layers (avgpool, full connected, softmax) are designed for classification and pre-training, and thus being removed when it is adopted for YOLO feature extraction. Darknet-53 contains 23 residual units. Each residual unit contains one 3 ×3 and

one 1 ×1 convolutional layer. At the end of each residual unit, an element-wise addition is performed between the input and output vectors. Batch Normalization (BN) is used after each convolutional layer followed by the Leaky ReLU activation function. Also, the down-sampling step is performed in five separate convolutional layers with a stride of 2. The backbone Darknet-53 is pre-trained on ImageNet dataset and performs on par with state-of-the-art classifiers but with less computational cost (classification accuracy is 0.1% higher and speed is 1.5 times faster than the ResNet-101 according to [original paper]).

The head subnet of YOLO v3 is built on top of the backbone to perform classification and bounding box regression and then output the predicted results. The head subnet adopts a Feature Pyramid Network (FPN) to detect objects at three different scales. FPN augments a standard convolutional network with a top-down pathway and lateral connections such that the network efficiently constructs a rich, multi-scale feature pyramid from a single resolution input image. Each level of the pyramid can be used for detecting objects at a different scale. The lower resolution feature maps have larger strides that leads to a very coarse representation of the input image, which is assigned for large object detection. While the higher resolution feature maps have more fine-grained features and is used for small object detection. YOLOv3 builds FPN on top of the backbone architecture and constructs a pyramid with down sampling strides 32, 16, and 8. It uses concatenation to perform the merging step in lateral connections instead of element-wise addition.

B) Prediction feature map

YOLO v3 makes predictions at three different scales (i.e. different grid resolutions) for image size NxN: $\frac{N}{32} \times \frac{N}{32}$, $\frac{N}{16} \times \frac{N}{16}$, $\frac{N}{8} \times \frac{N}{8}$, corresponding three down sampling strides 32, 16, and 8, respectively. For example, if the image size is 416x416, the three grid scales are 13x13, 26x26, 52x52. For each grid scale, YOLO v3 makes 3 predictions per grid. Each prediction composes of a bounding box, an objectness and 80 class scores.

To determine the anchor boxes, YOLOv3 applies k-means cluster and select 9 clusters. For COCO dataset, the width and height of the anchors 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 scale. The first three smaller anchor boxes are assigned to one group for the finer grid scale (e.g. 52x52), The middle three ancho boxes are used for the middle fine grid scale (e.g. 26x26), and the last three anchor boxes are used for the coarse grid scale (e.g. 13x13).

YOLOv3 uses multi-label classification. For example, the output labels may be “pedestrian” and “child” which are not non-exclusive. YOLOv3 replaces the softmax function with independent logistic classifiers to calculate the likeliness of the input belongs to a specific label. Instead of using mean square error in calculating the classification loss, YOLO v3 uses binary cross-entropy loss for the class predictions during training.

The inference to obtain the final bounding boxes for detected objects is similar to YOLO v2, except that all prediction bounding boxes at three scale output tensors need to be considered.

C) Loss function

Well-trained YOLO v3 models are available in open resources for detecting pre-defined general 80 classes. However, there are some applications where we need to define new and unusual classes. For instance, we want to detect persons at conference who raised their hands [8]. To

automatically make a notation of a chess game, we need to detect and recognize all the chess pieces on the board [9]. In these applications, we have to train the custom YOLO v3 models based on our own dataset. The basic idea of training a custom YOLO v3 is similar to general YOLO v3 training: starting with pre-trained Darknet and then tuning the entire network using the custom dataset.

The key to successfully training the network is to calculate losses. As we mentioned, at each scale we define three anchor boxes for each grid, and we need to pick the anchor box that has the highest IOU with the target box. After converting the truth (label) bounding box parameters to the format of prediction feature map (see Fig.21), we can then calculate the loss. As we discussed earlier, the loss function has multiple parts:

1) Bounding box coordinates error and dimension error that is represented using mean square error.

2) Objectness error which is confidence score of whether there is an object or not. When there is an object, we want the score equals to IOU, and when there is no object, we want to the score to be zero. This is also mean square error.

3) Classification error, which uses cross entropy loss.

Thus, the loss function can be calculated similarly to equation (6).

D) Comparison between YOLO2 and YOLO3

In general, YOLO v3 has a more powerful detection ability while keeping the same speed as YOLO v2. A deep comparison can be found by reading their original papers. In brief, YOLO v3 can likely detect more objects with a variety of sizes due to its three-grid-scale prediction and a powerful feature extractor (Darknet-53, instead of Darknet-19). Fig.25 shows an example of the results.

Fig.25 Results of YOLO2 and YOLO3: (left) original image,
(middle) YOLO2: 5 persons detected, (right) YOLO3: 9 persons and 1 handbag detected

Summary

This chapter describes a popular object detection algorithm – YOLO and its different versions. Their architectures have been presented in an evolved order so that we can understand how the algorithm has been improved through different versions. Before we discussed the YOLO algorithm, we had addressed the basic concepts: localization, sliding window and its ConvNet implementation, anchor box. 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. The loss function has been described for training purpose. It is very beneficial for one to read the original papers [1][2]5[6], and other posts [4] [10] [11] online for more detailed information. In the next chapter, to make it concrete, we will implement YOLO v3 from the scratch through a comprehensive tutorial.

References

[1] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. CoRR, abs/1312.6229, 2013

[2] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. *arXiv preprint arXiv:1506.02640*, 2015.

[3] M. Everingham, S. M. A. Eslami, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The pascal visual object classes challenge: A retrospective. International Journal of Computer Vision, 111(1):98–136, Jan. 2015. 2

[4] Carol Hsin, "Yolo Object Detectors: Final Layers and Loss Functions", at https://medium.com/oracledevs/final-layers-and-loss-functions-of-single-stage-detectors-part-1-4abbfa9aa71c

[5] J. Redmon and A. Farhadi. Yolo9000: Better, faster, stronger. In Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on, pages 6517–6525. IEEE, 2017.

[6] Redmon, J. & Farhadi, A. (2018), Yolov3: an incremental improvement, arXiv preprint arXiv:1804.02767.

[7] https://arxiv.org/ftp/arxiv/papers/1812/1812.10590.pdf

[8] http://leiluoray.com/2018/11/10/Implementing-YOLOV3-Using-PyTorch/

[9]https://towardsdatascience.com/training-a-yolov3-object-detection-model-with-a-custom-dataset-4981fa480af0

[10]Ayoosh Kathuria, 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/

[11] https://github.com/pjreddie/darknet