In object detection, we are returning a *bounding box* (the smallest possible box) containing the object.

In some ways it is similar to segmentation, but while segmentation can be understood as an image to image process and thus the *loss function* can be understood easily (it is the distance between the real segmentation and the segmentation guessed by the network), the same cannot be said about the object detection approach.

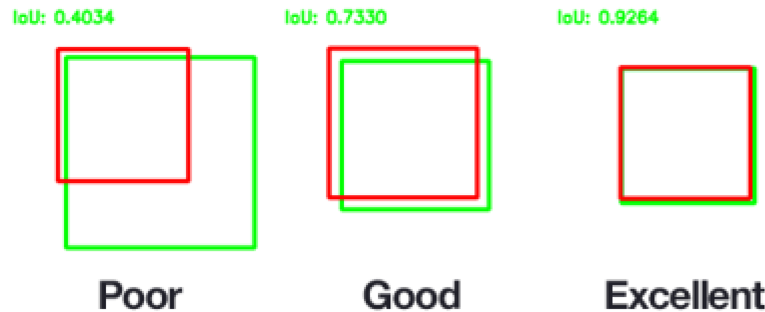
Pros & cons of this approach

- **pro**: no need to strive about borders
- **cons**:
    - multiple outputs of unknown number (the result is not of a fixed dimension)
    - difficult to train end-to-end
    - no evident loss function

How can we compute a loss function for this type of approach?

Intersection over Union - Quality indicator for Bounding Boxes Typically, the quality of each individual bounding box is evaluated vs. the corresponding ground truth using *Intersection over Union*:

$$IoU(A, B) = \frac{|A \cap B|}{|A \cup B|}$$



Essentially, we measure how good is a prediction by checking how much the 2 boxes stack up/ how close they are.

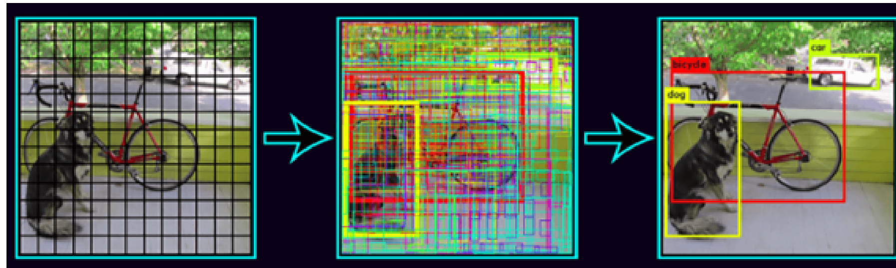
Then, these bounding boxes be summed up for all detections, and suitably combined with classification errors.

Deep Object Detection Approaches

There are 2 main approaches: - **Region Proposals methods** (R-CNN, Fast R-CNN, Faster R-CNN). Region Proposals are usually extracted via *Selective*

Search algorithms, aimed to identify possible locations of interest. These algorithms *typically exploit the texture and structure of the image*, and are object independent. - Detectron2 is a pytorch library developed by Facebook AI Research (FAIR) to support rapid implementation and evaluation of novel computer vision research. - **Single shots methods** (Yolo, SSD, Retina-net, FPN). We shall mostly focus on these *really fast* techniques. - In this kind of methods, we are trying to do everything on a single pass, meaning that we're trying to identify the boxes, and then to classify the content of the box. - Especially suited for real-time applications.

YOLO's architecture

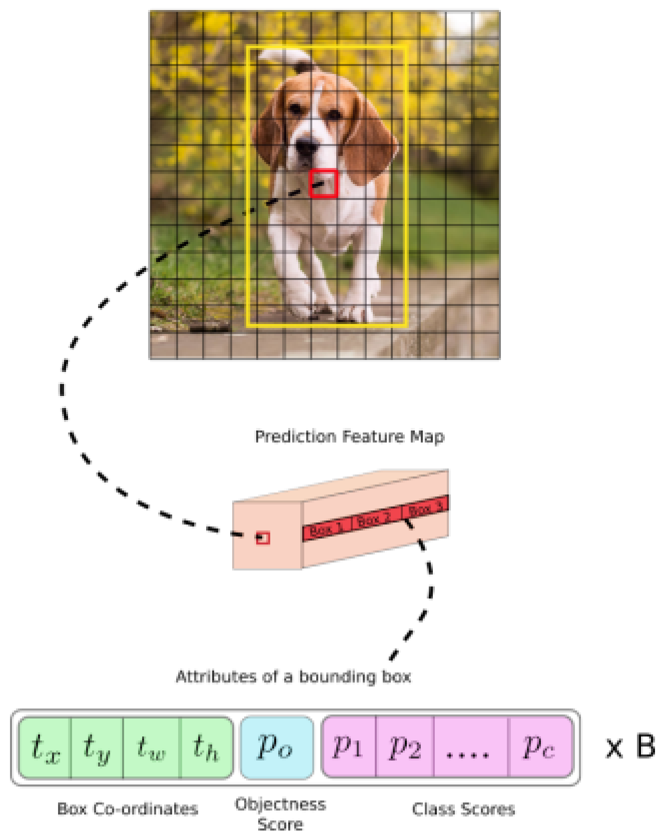


Yolo is a *Fully Convolutional Network*. The input is *progressively downsampled* by a factor $2^5 = 32$. This is done in order to *increase the receptive field of our neurons* (each neuron at the end musts see a sufficiently large portion of the image). For instance, an input image of dimension 416x416 would be reduced to a grid of neurons of dimension 13x13, which is *the feature map.*

How do we train this network? Detection of an object may concern all neurons inside the bounding box. So, who's in charge for detection (who's supposed to recognize this objects, and thus should be trained)?

In YOLO, a *single neuron* is responsible for *detection*: the ==one whose grid-cell contains the *center* of the bounding box==. This neuron makes a *finite number of predictions* (e.g. 3). - We don't care what all the other neurons predict, in fact they get *masked* in the loss function.

Shape of each box We have 13x13 neurons in the feature map. - Depth-wise, we have $(B \times (5 + C))$ *entries*, where B represents the *number of bounding boxes* each cell can predict (say, 3), and C is the number of *different object categories.* - Each bounding box has $5 + C$ attributes, which describe the *center coordinates* (2), the *dimensions* (2), the *objectness score* (1) and C class confidences (1 for each prediction, usually is 3).



Anchor Boxes Trying to directly predict width and the height of the bounding box leads to *unstable gradients* during *training*. Most of the modern object detectors predict log-space affine *transforms* for *pre-defined default bounding boxes* called **anchors**. Then, these transforms are applied to the anchor boxes to obtain the prediction. YOLO v3 has three anchors, which result in prediction of three bounding boxes per cell. The bounding box responsible for detecting the object is one whose anchor has the highest IoU with the *ground truth box*.

[to add: slides that he skipped...]

YOLO's Loss function

The loss consists of two parts, the **localization loss** for *bounding box offset prediction* and the **classification loss** for *conditional class probabilities*. Since YOLO is a single pass method, the final loss function should compute both of this parts in a single computation. As usual, we shall use v to denote a true value, and $\hat{v}$ to denote the corresponding predicted one.

Localization loss The localization loss is:

$$\mathcal{L}_{loc} = \sum_{i=0}^{S^2}\sum_{j=0}^{B} 1_{ij}^{obj}[(x_i-\hat{x}_i)^2+(y_i-\hat{y}_i)^2+(\sqrt{w_i}-\sqrt{\hat{w}_i})^2+(\sqrt{}$$

where i ranges over cells, and j over bounding boxes. - 1_{ij}^{obj} is a *delta function* indicating whether the *j-th bounding box of the cell i is responsible for the object prediction.* - Essentially, it is a mapping function containing 0 everywhere except for the neuron that is supposed to do the localization. This essentially masks the predictions of the other neurons.

Classification loss The classification loss is the sum of two components, relative to the *objectness confidence* and the *actual classification*:

$$\mathcal{L}_{cls} = \sum_{i=0}^{S^2}\sum_{j=0}^{B}(1_{ij}^{obj} + \lambda_{noobj}(1 - 1_{ij}^{obj}))(C_{ij} - \hat{C}_{ij})^2$$

$$+ \sum_{i=0}^{S^2}\sum_{c\in C} 1_{i}^{obj}(p_i(c) - \hat{p}_i(c))^2$$

λ_{noobj} is a configurable parameter meant to down-weigth the loss contributed by "background" cells containing no objects. This is important because they are a large majority.

Final result The whole loss is:

$$\mathcal{L} = \lambda_{coord}\mathcal{L}_{loc} + \mathcal{L}_{cls}$$

λ_{coord} is an additional parameter, balancing the contribution between L_{loc} and L_{cls}. In YOLO, $\lambda_{coord} = 5$ and $\lambda_{noobj} = 0.5$.

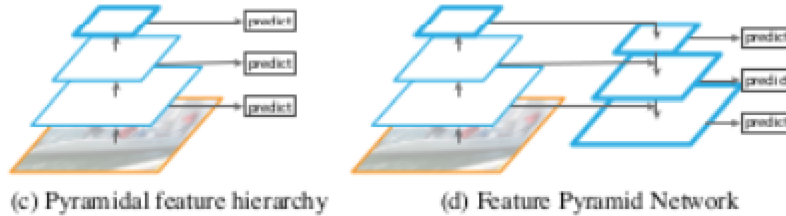
Multi scale processing

Here's an overview of image processing techniques for object detection throughout history.

(a) Featurized image pyramid (b) Single feature map

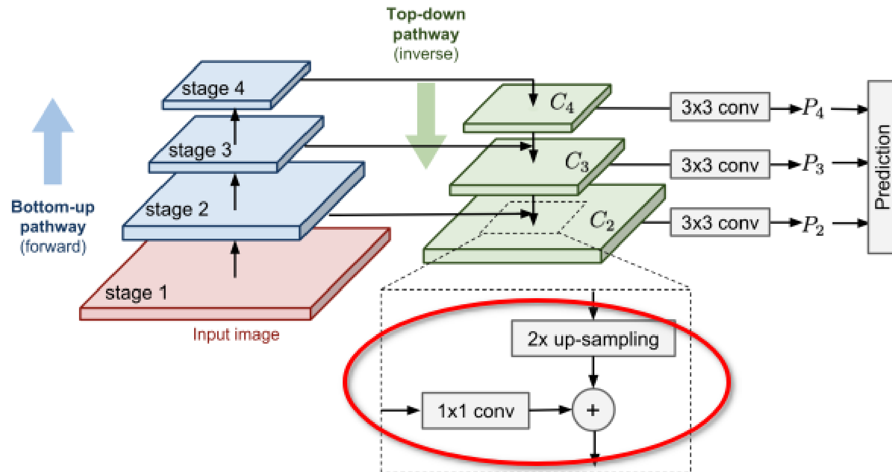
- The older approach to object detection from the 2010s used a **featurized image pyramid**. With this approach, features are computed on each of the image scales independently, which is *slow*. - Essentially, images were scaled and rescaled multiple times in order to find the important features of the image.

- First systems for fast object detection (like YOLO v1) opted to use only higher level features at the *smallest scale* (**single feature map**). This usually *compromises detection of small objects.*



- An alternative (Single Shot Detector - SSD) is to reuse the *pyramidal feature hierarchy* computed by a ConvNet as if it were a *featurized image pyramid.*
- Modern Systems (FPN, RetinaNet, YOLOv3) recombine features along a **backward pathway**. This is as fast as (b) and (c), but more accurate.

In the figures, feature maps are indicated by blue outlines and thicker outlines denote semantically stronger features.

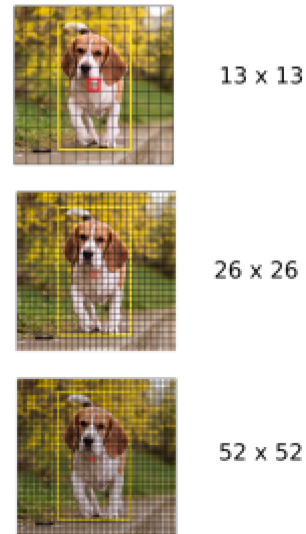


Featurized Image Pyramid

- Bottom-up pathway is the normal feedforward computation. - *Top-down* pathway goes in the inverse direction, adding coarse but *semantically stronger feature maps* back into the previous pyramid levels of a larger size via lateral connections. - First, the higher-level features are *spatially upsampled.* - The feature map coming from the Bottom-up pathway undergoes *channels reduction* via a 1x1 conv layer. - Finally, these two feature maps are *merged* (by element-wise addition, or concatenation).

Non Maximum Suppression

This is final phase of the YOLO algorithm. Essentially, we have to consider that we have a huge amount of predictions, since at each feature map, each neuron makes a prediction. YOLOv3 predicts feature maps at scales 13, 26 and 52.



For example, if we have a situation like this:
At the end, we have $((13 \times 13) + (26 \times 26) + (52 \times 52)) \times 3 = 10647$ bounding boxes, each one of dimension 85 (4 coordinates, 1 confidence, 80 class probabilities). How can we reduce this number to the few bounding boxes we expect?

These operations are done algorithmically, and they consist in - **Thresholding by Object Confidence**: first, we filter boxes based on their objectness score. Generally, boxes having scores *below a threshold are ignored.* - **Non Maximum Suppression**: NMS addresses the problem of *multiple detections of the same image*, corresponding to different anchors, adjacent cells in maps.

NMS outline

- Divide the bounding boxes BB according to the predicted class c (creating a list for each class).
- Each list BB_c is processed separately
- Order BB_c according to the object confidence.
- Initialize `TruePredictions` to an empty list.
- while BB_c is not empy:
  - pop the first element p from BB_c

  - add p to `TruePredictions`
  - remove from BB_c all elements with an IoU with $p > th$
- return `TruePredictions`

Essentially, it ignores all overlapping BBs and keeps only the best ones.