Part 06 - Object Detection

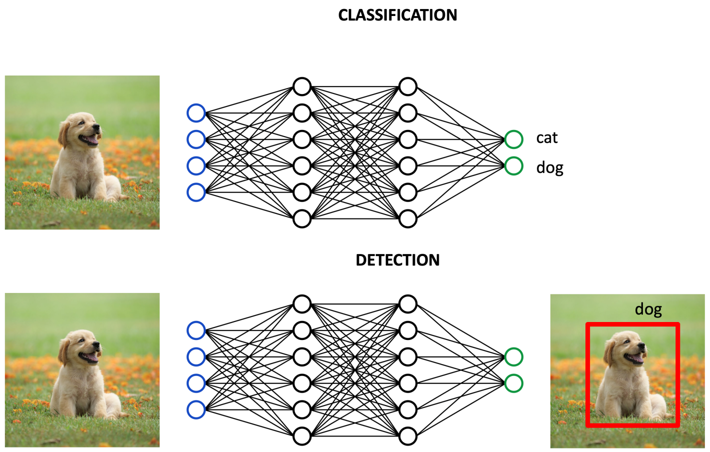
Maura Pintor (maura.pintor@unica.it)

Credits for this lecture go primarily to this series of video tutorials

Localization vs Detection vs Segmentation

- **Localization**: Find what and where a single object exists in an image
  - output: a box with a label
- **Detection**: Find what and where multiple objects exist in an image
  - output: multiple boxes with labels
- **Segmentation**: Find which pixels contain a specific object in an image
  - output: pixel-wise labels

Object detection outputs



In classification, the prediction is given by the highest score: $\text{argmax}(c_1, \ldots c_n)$

In localization, the prediction has the format: $(c_1, \ldots c_N, p, x_1, y_1, x_2, y_2)$

- $(c_1, \ldots c_N)$ are the classes
- p counts for the likelihood of the presence of an object inside the bounding box
- (x_1, y_1, x_2, y_2) are the box coordinates

Object detection outputs

We have two options for defining the **bounding boxes (BBs)**:

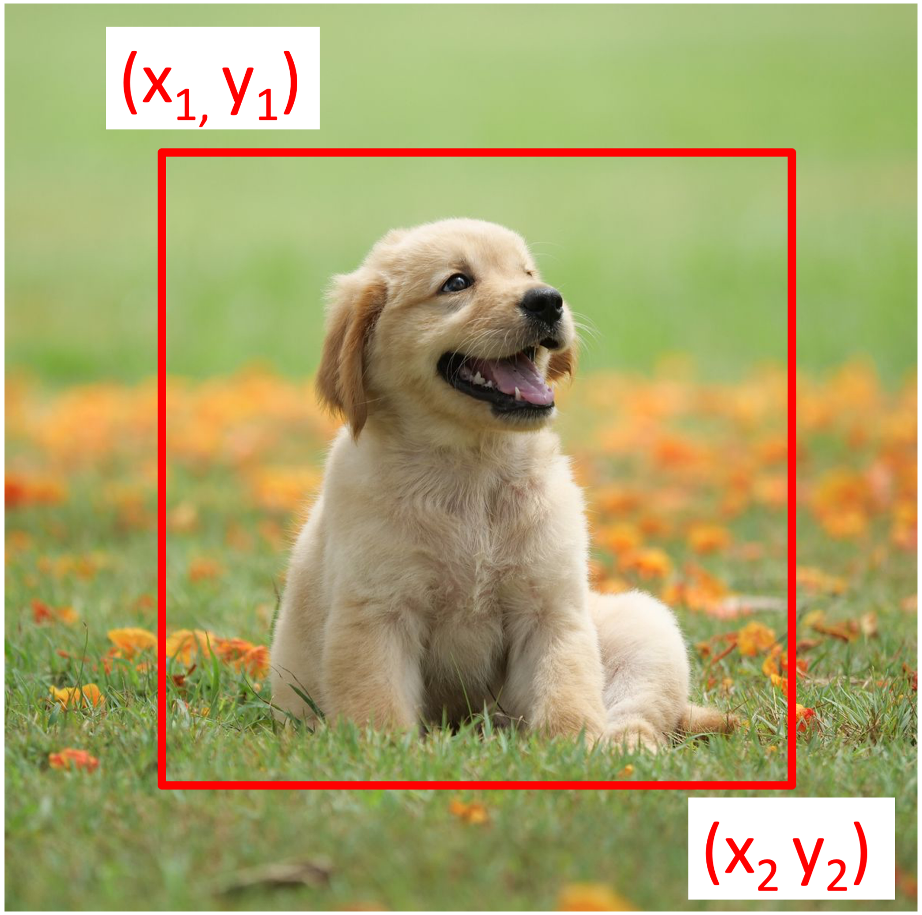
- (x_1, y_1, x_2, y_2)
- (x_1, y_1, w, h)

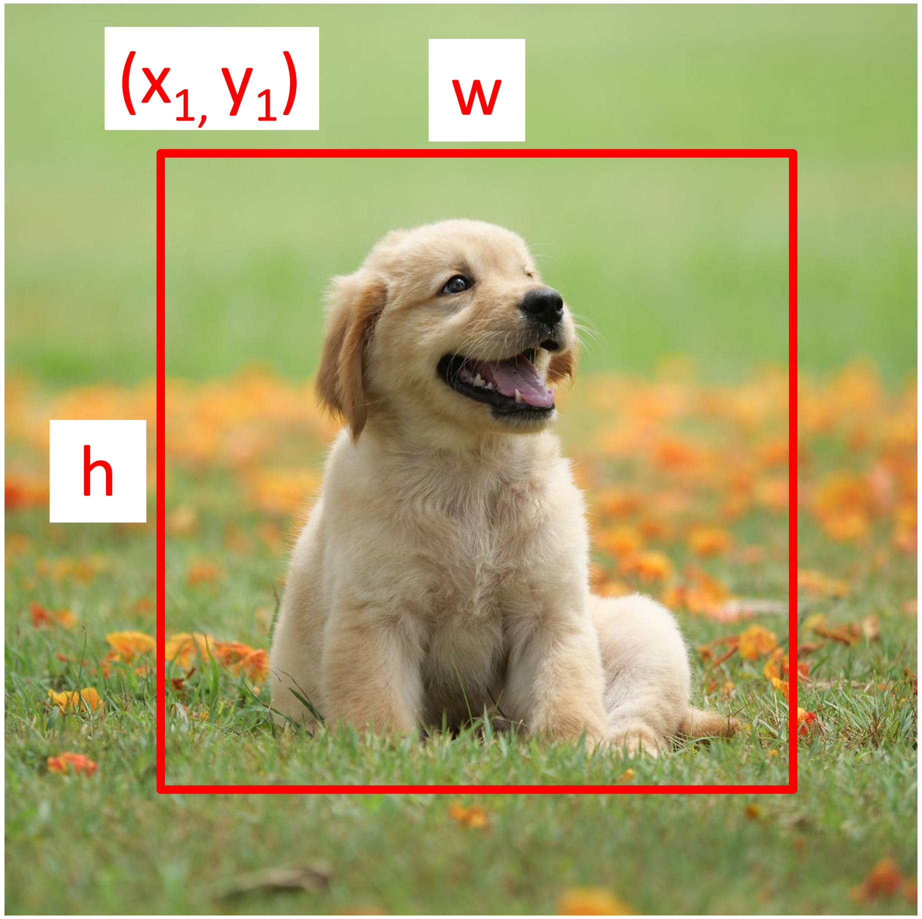
For now, we will focus on the first representation, but it's easy to change from one to the other:

$$w = x_2 - x_1; h = y_2 - y_1$$

$$x_2 = x_1 + w; y_2 = y_1 + h$$

Object detection outputs





Detecting objects

How can we train the model to learn all objectives?

1. correct labels of the objects
2. correct box coordinates
3. generalize for multiple objects in a single image

Detecting objects

Sliding window: define a BB and check if there is an object. Then slide the window and check again. If there is an object, we perform classification.

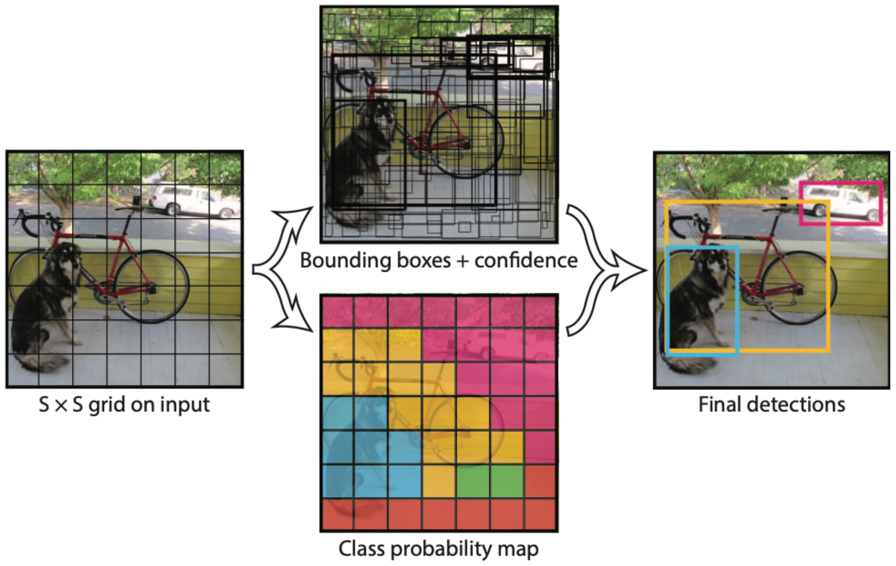
- We can implement the sliding window approach with a ConvNet

However, with this approach, we can have many BB for the same object!

- There are tricks to remove redundant BBs (it's called *Non max suppression*)

YOLO: You Only Look Once

1. divide image in $S \times S$ grid
2. each cell is responsible to predict bounding box (centered in the cell) and confidence + class probability for the cell
3. final detection is a filtered version of the combination of the outputs



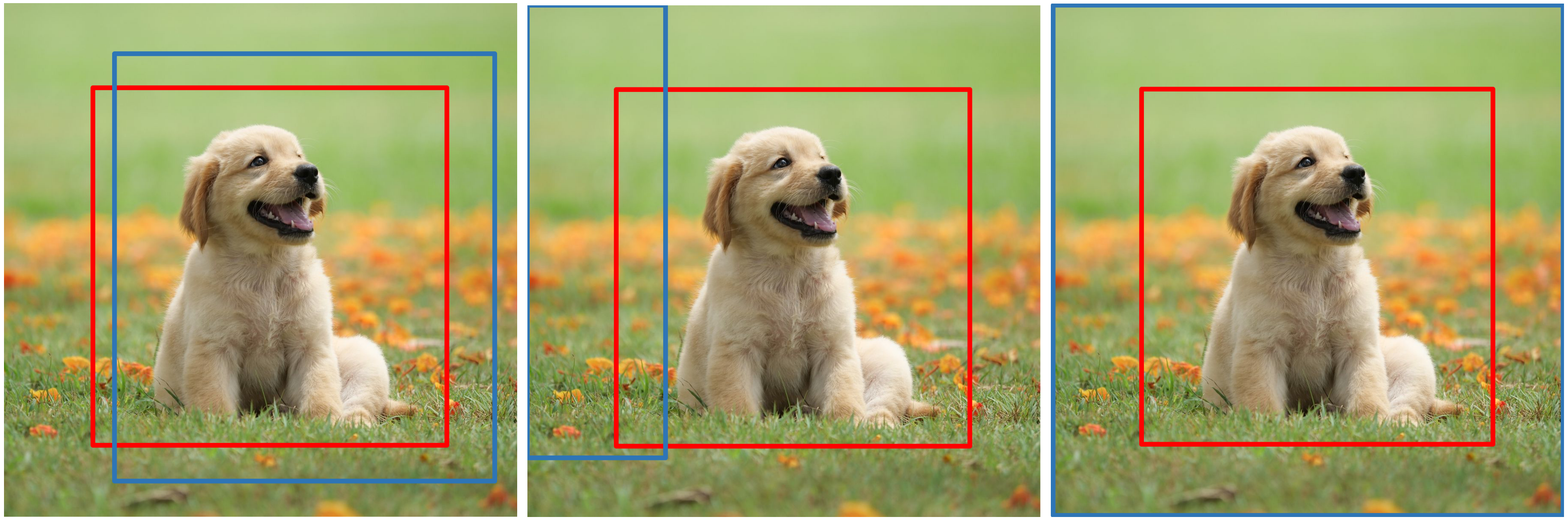
Bounding Boxes

How do we measure how good a BB is?

Our goal: compare prediction vs ground truth (GT)

- We want to maximize the area of GT covered by the prediction
- but it's easier to cover the whole image, at this point we have maximum area of GT covered with no effort
- we have to penalize the area of prediction that is not part of the ground truth

Bounding Boxes



Bounding Boxes

Intersection over union (IoU): We want to encourage small regions that correspond to the true region and penalize big regions that don't belong to the correct region

$$\text{IoU} = \frac{\text{intersection}}{\text{union}} = \frac{I}{U}$$

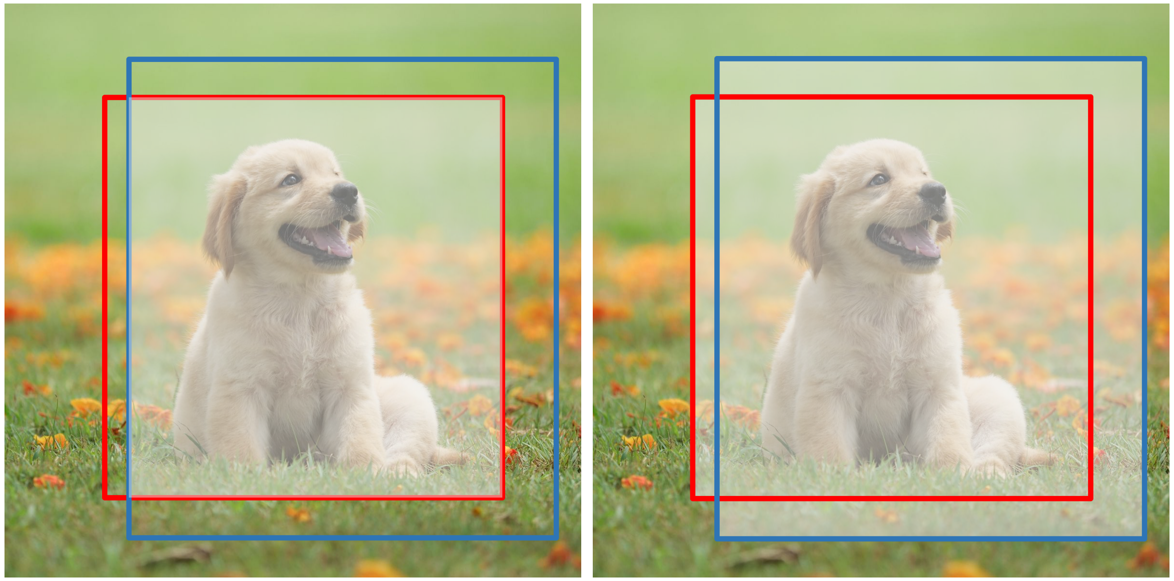
If I is big we have a big *intersection* of the two areas, and the IoU increases.

If GT is the same as the prediction, the *union* U is minimal.

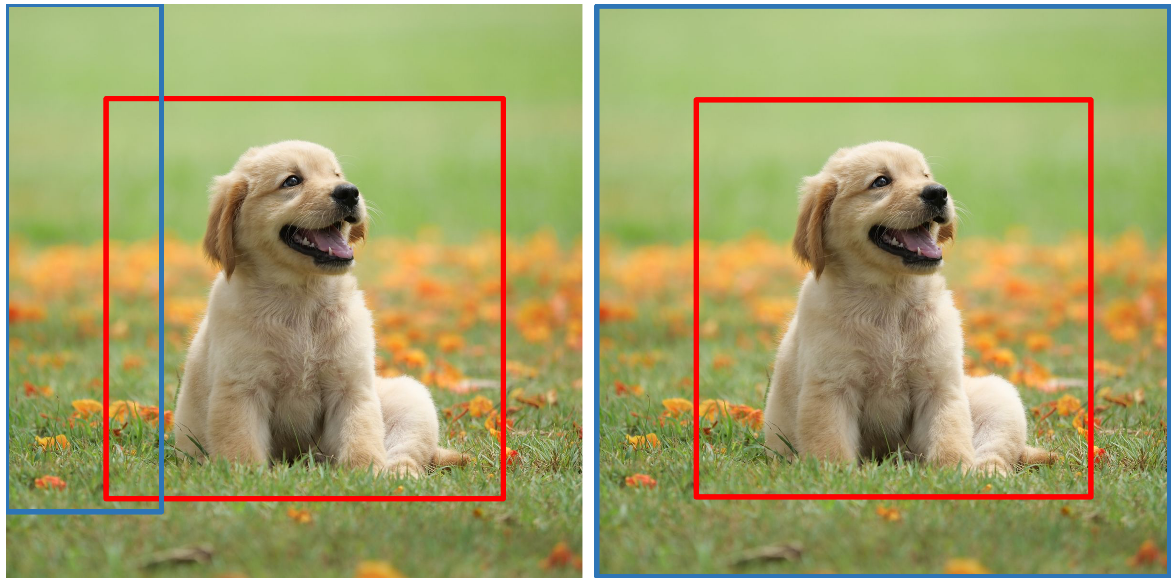
If IoU=1 we have the perfect match between ground truth and prediction, as $I = U$.

IoU is always ≤ 1 and equal to 1 only when it's perfect (in general $\text{IoU} > 0.5$ is considered "decent")

Bounding Boxes

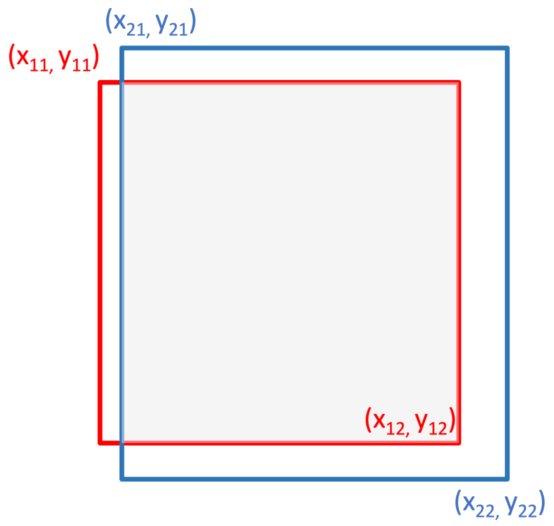


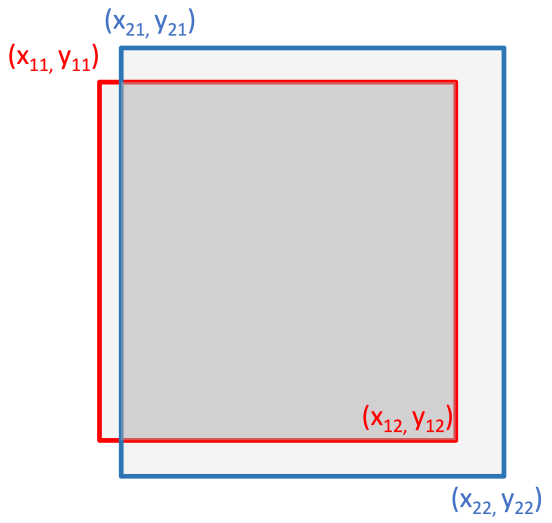
Bounding Boxes



Implementing IoU

Intersection from two lists of numbers





$$B_1 = [x_{11}, y_{11}, x_{12}, y_{12}]$$

$$B_2 = [x_{21}, y_{21}, x_{22}, y_{22}]$$

$$I = [\max(x_{11}, x_{21}), \max(y_{11}, y_{21}), \min(x_{12}, x_{22}), \min(y_{12}, y_{22})],$$

Implementing IoU

Let's implement a simple version (for a single box and prediction):

```python
import torch

def intersection_over_union(boxes_pred, boxes_true):
    box1_x1, box1_y1, box1_x2, box1_y2 = boxes_pred
    box2_x1, box2_y1, box2_x2, box2_y2 = boxes_true

    # compute intersection
    i_x1 = torch.max(box1_x1, box2_x1)
    i_y1 = torch.max(box1_y1, box2_y1)
    i_x2 = torch.min(box1_x2, box2_x2)
    i_y2 = torch.min(box1_y2, box2_y2)

    # the clamp takes care of the case for which they don't intersect
    intersection = (i_x2 - i_x1).clamp(0) * (i_y2 - i_y1).clamp(0)

    # compute union
    box1_area = (box1_x2 - box1_x1) * (box1_y2 - box1_y1)
    box2_area = (box2_x2 - box2_x1) * (box2_y2 - box2_y1)

    union = box1_area + box2_area - intersection

    return intersection / (union + 1e-6)
```

Implementing IoU

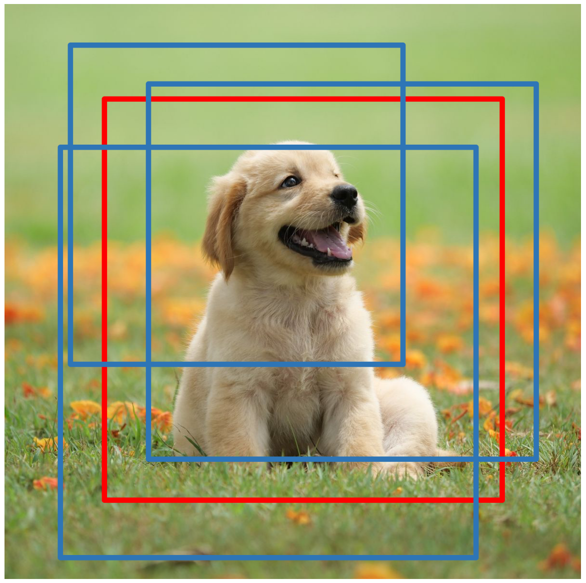
And then let's try it:

```
box_1 = torch.tensor([0, 0, 1, 1])
box_2 = torch.tensor([0.5, 0.5, 1, 1])

print(intersection_over_union(box_1, box_1))
print(intersection_over_union(box_1, box_2))
```

Implementing Non Max suppression

We have to cleanup BB predictions (if we have multiple overlapping BBs for the same object)



Implementing Non Max suppression

Each BB will have a probability indicating how likely it is that there is an obj in the BB

Non Max Suppression (NMS) takes only the highest scoring box, and removes the redundant boxes

NMS uses the IoU between each couple of BBs, if IoU is higher than a threshold it removes a box from the prediction

We have to keep boxes if they belong to different classes (there might be different objects close to each other)

Implementing Non Max suppression

So we have to implement the following steps:

For each class in $c_1, \ldots, c_N$:

- remove all BBs with $p < T_p$, where p is the probability of object and T_p is a decided threshold
- consider the largest probability box
    - remove all other boxes with $IoU > T_{\text{IoU}}$

Implementing Non Max suppression

```python
def non_max_suppression(bboxes, iou_threshold, prob_threshold):
    # bboxes = [[class, probability of BB, x1, y1, x2, y2], [], []]
    bboxes = [box for box in bboxes if box[1] > prob_threshold]
    bboxes = sorted(bboxes, key=lambda x: x[1], reverse=True)
    bboxes_after_nms = []
    while bboxes:
        chosen_box = bboxes.pop(0)
        bboxes = [
            box
            for box in bboxes
            if box[0] != chosen_box[0]
            or intersection_over_union(
                torch.tensor(chosen_box[2:]),
                torch.tensor(box[2:]),
            )
            < iou_threshold
        ]

        bboxes_after_nms.append(chosen_box)

    return bboxes_after_nms
```

Implementing Non Max suppression

Then let's build a test for our function:

```
bboxes = [
            [1, 1, 0.1, 0.45, 0.5, 0.7],
            [1, 0.9, 0.1, 0.45, 0.45, 0.6],  # same as the first
            [2, 0.9, 0.1, 0.45, 0.45, 0.6],  # same as first but diff class
            [1, 0.7, 0.25, 0.35, 0.3, 0.1],
            [1, 0.05, 0.1, 0.1, 0.1, 0.1],  # low probability
        ]

print(non_max_suppression(bboxes, prob_threshold=0.2, iou_threshold=0.5))
```

The expected output should filter out only the second (it's the same as the first but with lower probability) and the last (small probability of object).

Implementing Non Max suppression

Let's also try it for a real image. First, let's get an image from the web:

```
import urllib
import PIL
import matplotlib.pyplot as plt

image_url = 'https://hips.hearstapps.com/ghk.h-cdn.co/assets/16/08/gettyimages-5

urllib.request.urlretrieve(image_url, "example.png")
img = PIL.Image.open("example.png")
plt.imshow(img)
```

Implementing Non Max suppression

Then, let's create a few bounding boxes:

```
import cv2
import numpy as np

def draw_bbox(img, bbox, colors):
    x1, y1, x2, y2 = bbox
    rec = cv2.rectangle(np.array(img), (x1, y1), (x2, y2), colors, 3)
    return rec


ground_truth = (200, 100, 600, 800)
preds = ((0, 0.9, 200, 200, 600, 800),
         (0, 0.6, 200, 220, 600, 800),
         (0, 0.95, 190, 200, 700, 800),
         )


for bbox in preds:
    img = draw_bbox(img, bbox[2:], (0, 0, 255))

img = draw_bbox(img, ground_truth, (255, 0, 0))

plt.imshow(img)
```

Implementing Non Max suppression

Then, let's apply NMS:

```
boxes = non_max_suppression(preds, iou_threshold=0.4, prob_threshold=0.7)

print("number of bboxes before nms: ", len(preds))
print("number of bboxes after nms: ", len(boxes))

image = img

for bbox in boxes:
    image = draw_bbox(image, bbox[2:], (0, 0, 255))

image = draw_bbox(image, ground_truth, (255, 0, 0))

plt.imshow(image);
```

Now, we need a way of understanding if the model is predicting the right BBs for each image...

Mean Average Precision (mAP)

Most used metric to evaluate object detection models

It's usually represented as a string:

```
mAP@0.5:0.05:0.95
```

Don't worry if you are not able to read it now. We will be able to understand it soon.

Mean Average Precision (mAP)

Goal: have one BB for each GT bounding box, with the correct class label

- TP = True Positive = one BB is identified correctly (box and class label)
- TN = True Negative = no BB is identified and there is no GT object in the image
- FP = False Positive = one BB is identified but there is no GT object
- FN = False Negative = no BB is identified but there is a GT object in the image

precision = $\frac{\text{TP}}{(\text{TP}+\text{FP})}$

recall = $\frac{\text{TP}}{(\text{TP}+\text{FN})}$

Implementing Mean Average Precision (mAP)

It's a score averaged over multiple things, so we have to implement it with nested for loops...

Steps:

1. get all bounding box predictions on our test set
2. sort by descending confidence score
3. compute precision and recall (for given thresholds) as we go through all outputs
4. repeat for different IoU thresholds, average for a range of values
5. compute the area of the curve
6. repeat for each class and average

Implementing Mean Average Precision (mAP)

Let's move to the Colab for this one, the implementation is not complicated but long to put in only one slide...

- Notebook with the code used in these slides

We then have all the pieces to evaluate our image detection models.

Implementing YOLO v1

For this implementation, we will use a repository with the code already written, but we will inspect the most important parts.

We will not train a YOLO v1 model (it takes some time), but we can use a pretrained version and use it for predictions.

First, let's see how the model is implemented.

Implementing YOLO v1

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

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

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

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

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

Implementing YOLO v1

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

box coordinates with the identity to count if the cell is responsible for the object

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

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

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

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

Implementing YOLO v1

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

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

accounts for boxes width and height

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

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

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

Implementing YOLO v1

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

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

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

probability of a box

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

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

Implementing YOLO v1

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

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

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

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

accounts for cells without a box

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

Implementing YOLO v1

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

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

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

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

forces to learn the right class predictions

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

Implementing YOLO v1

Parts of the loss function (all parts are a squared error with the target):

- the first term is the box coordinates with identity (if there is a box in the cell and if the cell is responsible == highest IoU for the target)
- the second term is for the width and height of the cells. The square roots make the very large BB be penalized less.
- the third term regulates the probability of a box to contain an existing object (the one with the highest IoU with the target)
- the fourth term counts for cells without an object
- the last term is for predicting the rigth classes

Let's now move to the repository and check out the implementation: https://github.com/maurapintor/yolo_implementation

End of part 6

Summary:

- Object detection fundamentals
- Evaluating object detectors (metrics)
- Training object detectors (loss function)
- Implementation and usage

End of part 6

In the next chapter:

- Running experiments - good practices and tools

- Notebook with the code used in these slides
- Repository with the code used in these slides

Maura Pintor (maura.pintor@unica.it)