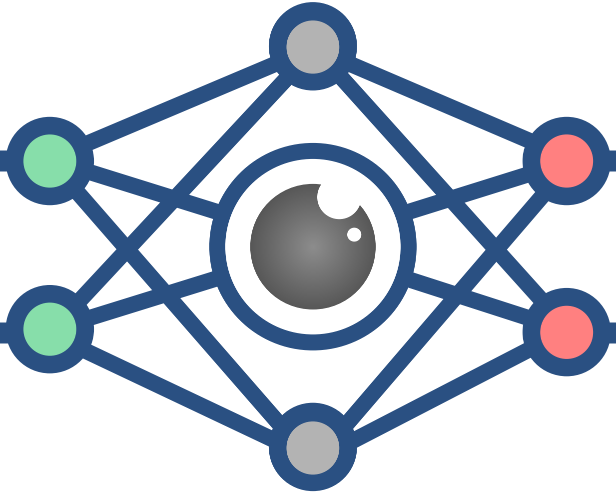
CS3485

Deep Learning for Computer Vision



Lec 13: Intro to Object Detection

(Tentative) Lecture Roadmap

Basics of Deep Learning

- Intro to Computer Vision
- Linear Classifiers and Perceptron
- Multilayer Perceptron
- Optimization in Deep Learning
- Pytorch I – MLPs
- Pytorch II – Images and Regularization
- Convolutional Neural Networks
- Data Augmentation and Deep CNNs
- Transfer Learning and Residual Nets
- Inception Net and what CNNs learn
- Adversarial Examples and Self-supervision

Deep Learning and Computer Vision in Practice

- Intro to MLOps
- Intro to Object Detection
- Fast Object Detection
- Intro to Image Segmentation
- Applications of Detection and Segmentation
- Autoencoders
- Image Generation with GANs
- Advanced GANs
- The Attention Mechanism
- Transformers and ChatGPT
- Image Generation by Prompt
- Misc. Topics

When Image Classification isn’t enough

- In the previous lectures, we learned about how to perform **image classification**.
- Now, imagine a self-driving car: for it, checking if the road it sees contains the images of vehicles, a sidewalk, and pedestrians isn’t enough.
- It is also important to identify where **those objects are located**!
- The various techniques for **object detection** we’ll study today and next time come in handy in such a scenario.

Object Localization

- To understand object detection we first need to see another vision task called **Object Localization**:

  Object Localization is the task of locating an instance of a particular object category in an image, typically by specifying a bounding box centered on the instance.

- The object's **bounding box (BB)** is a rectangle that tightly surrounds the object found and is the our desired output.
- BBs are specified by a tuple of four numbers:

  `(Cx, Cy, H, W)`

  where `Cx` and `Cy` are the BB (**normalized***) centers and `H` and `W` are its height and width, also **normalized**.

Bounding box

* Normalized in relation to the dimensions of the original image, so all these values are in *[0, 1]*.

Object Detection

- Object detection is then the joint work of image classification and object localization:

Object detection is the task of localizing instances of objects of a certain set of available classes within an image.

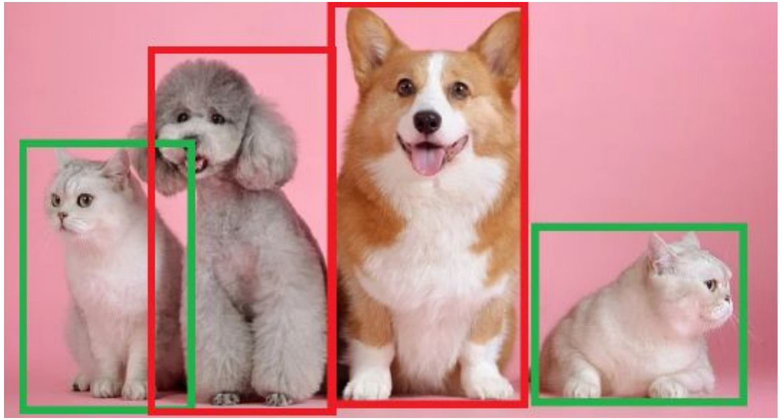
Object Classification

'cat'

Object Localization

(.4, .5, .5, .8)

Object Detection

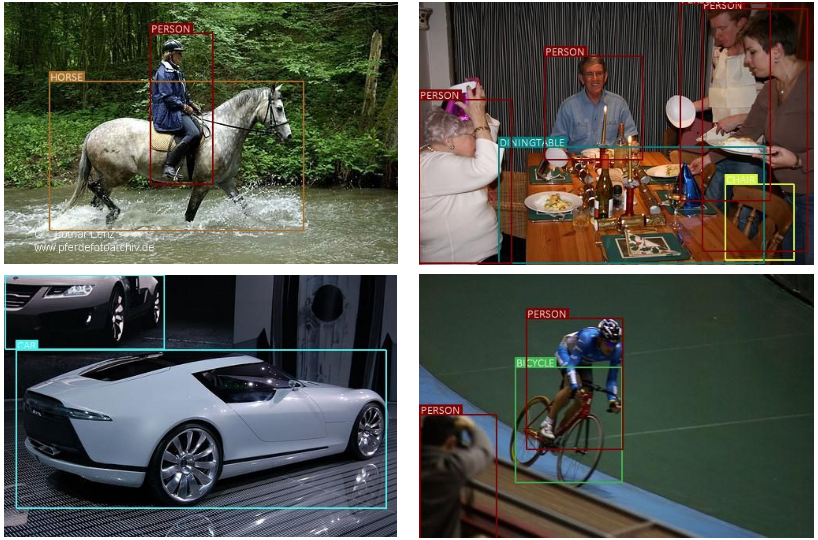


'cat', 'dog', 'dog', 'cat' + locations

Object Detection in Practice

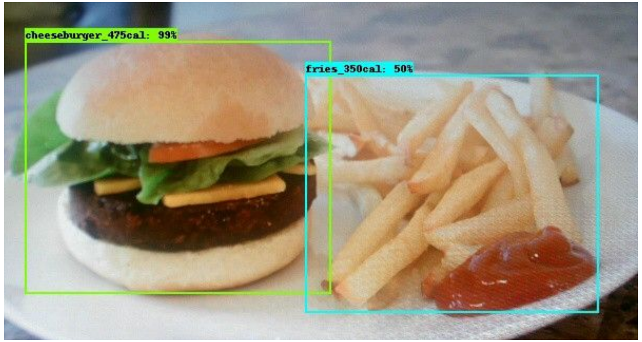
- Here are some examples of object detection outputs:

On Individual Images	On Videos



Applications of Object Detection

- Some of the **various** use cases leveraging object detection include the following:
  - **Surveillance**: This can be useful for recognizing intruders in places, count people in crowds, detecting hazardous situations, etc.
  - **Autonomous cars**: This can be helpful in recognizing the various objects present on the image of a road.
  - **Image search**: This can help identify the images containing an object of interest.
  - **Automotives IDing**: This can help in identifying a number plate within the image of a car.



Data involved in Object Detection

- As with image classification, we are doing supervised learning, so we **also need training data** in Deep Learning based Object Detection.
- This data is usually composed of at least a set of images with ids and a table that contains each class and bounding boxes vectors* (humanly annotated) about each image.

Example image (id: 3212)

Example bounding box table

Image id	Cx	Cy	W	H	Class
3212	0.3528	0.2741	0.3123	0.5859	bus
3212	0.6734	0.7932	0.0521	0.5270	person
3212	0.7589	0.7356	0.3257	0.4275	car
3212	0.7042	0.5278	0.0349	0.1290	person
3212	0.9531	0.6545	0.0790	0.5352	person

* Sometimes, the box info will come as the rectangles' `(xmin, xmax)` and `(ymin, ymax)`, instead of our `(Cx, Cy, H, W)` used here.

Object Proposals

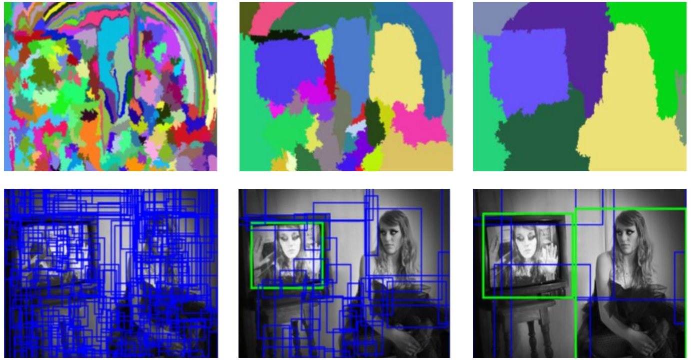
- After the network is trained (*more on it later*), we'll need to generate **object proposals** from a test image, from which objects' classes are to be **inferred**.
- To understand object proposals, imagine that the image of interest is grayscale and it contains a woman and a TV in the **foreground** and a wall in the **background**. Assume:
  - The colors in the background are usually lighter and don't change abruptly.
  - The colors in the foreground are darker and change very rapidly.
  - The pixels in each object are compact, i.e., each object is a sole blob of pixels instead of multiple separated blobs.
- It means that we can detect potential objects just from their **pixel colors and locations**.
- **Object proposals (**also called **region proposals)**, therefore, are regions of the image where the pixels are similar color-wise and close to one another.
- From each proposal we can draw a box (also called a **region of interest, RoI**), potentially containing an object in the image.

Object Proposals

- Unfortunately, the notions of similar and close are quite subjective subjective:
  - If we make them permissive (any similarity and closeness is enough to joining pixels together), we may end up with too many proposals, most of which are useless.
  - If we make them to strict, we may miss big objects (like the TV below) that are composed of smaller regions of different colors.
- The usual approach to solve this issue to **start with a very permissive set of proposals**, then **join them into larger regions** and repeat until a minimum amount of regions is found.

Original Image

Proposals generated after some iterations



Selective Search

- The process described before is called **selective search** and there is a library in Python conveniently called `selectivesearch` that implements this technique:

```
import selectivesearch
_, regions = selectivesearch.selective_search(img, scale=1.0, min_size=50)
```

  where `scale` corresponds to the permissiveness discussed before, `min_size` is the min. region size of each proposal in pixels and `regions` is a list with the BBs' info.
- In order to show an image with the bounding boxes of the proposals, we can use the function show from the `torch_snippets` library:

```
import torch_snippets
torch_snippets.show(img, bbs=list_of_bounding_boxes, texts=list_of_bb_classes)
```

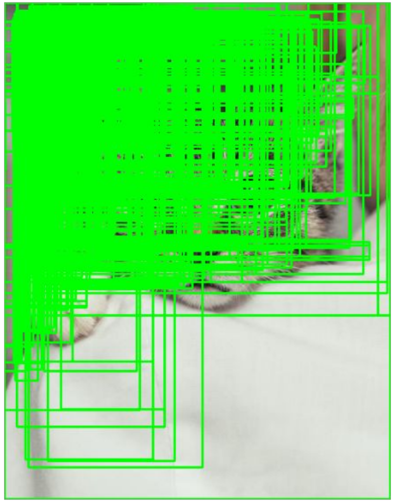
  where `bbs` is a list of tuples in the format `(xmin, xmax, ymin, ymax)` and `texts` is a list of strings that contain the label of each bounding box.

Selective Search

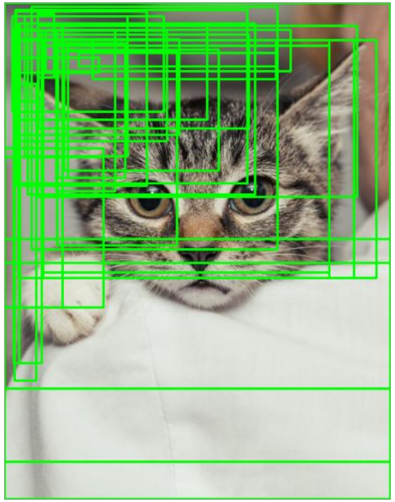
- Some selective search results from different values of `scale` and `min_size` (MS):

Original Image

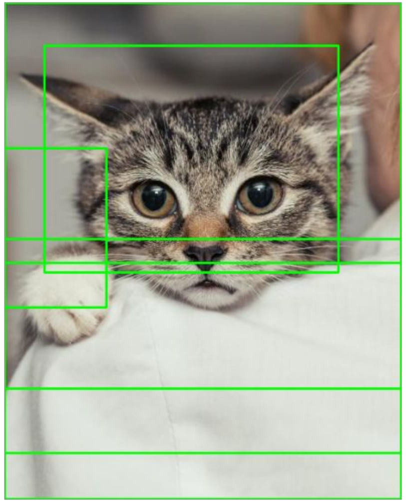
Low scale and MS



High scale and low MS



High scale and MS



- The goal is to hit a sweet spot by having enough proposals, not too many, not too few.

Exercise (*in pairs*)

Click here to open code in Colab

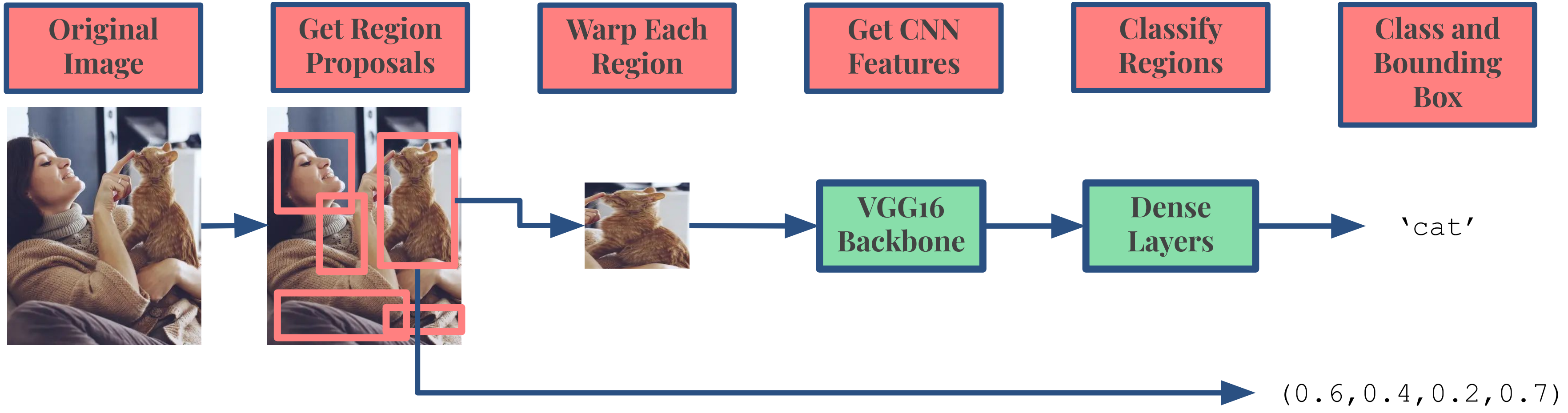
- The previous cat image's bounding boxes were generated using the following code:

```
!pip install selectivesearch torch_snippets # Don't forget to install them on Colab

import torchvision.io as io
import selectivesearch
import torch_snippets

img = io.read_image("cat.jpg").permute(1, 2, 0)
_, regions = selectivesearch.selective_search(img, scale=200, min_size=1000)
```

Now, download a new image from the internet of anything, generate its bounding boxes via selective search and show them `torch_snippets.show()`. Note that `regions` do not give a list of tuples corresponding the the BBs dimensions right away, it is infact a list of dictionaries. Explore what these dictionaries contain before you plot the BBs.

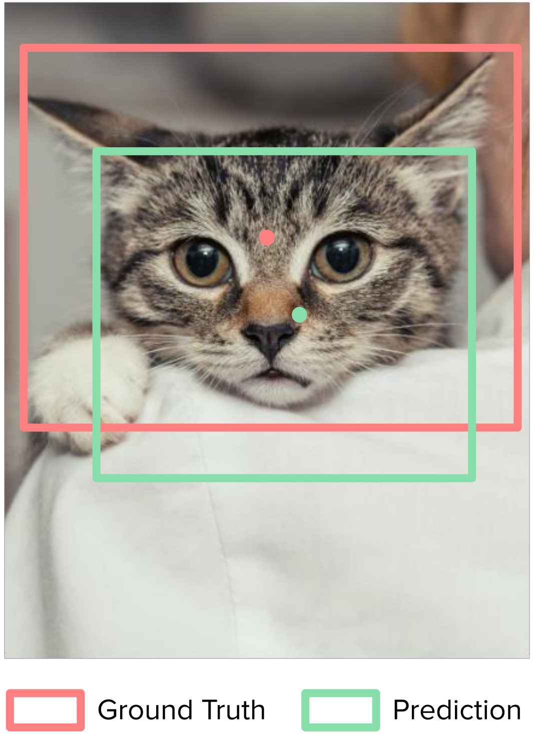
Naive Object Detection

- One way to perform detection is to classify each proposal using a pre-trained net (like VGG16) fine-tuned to the desired classes (here **we'd also add a class for background**).
- Then, our output would the each predicted class and each proposal location, along with the classification confidence that that region belongs the predicted class.



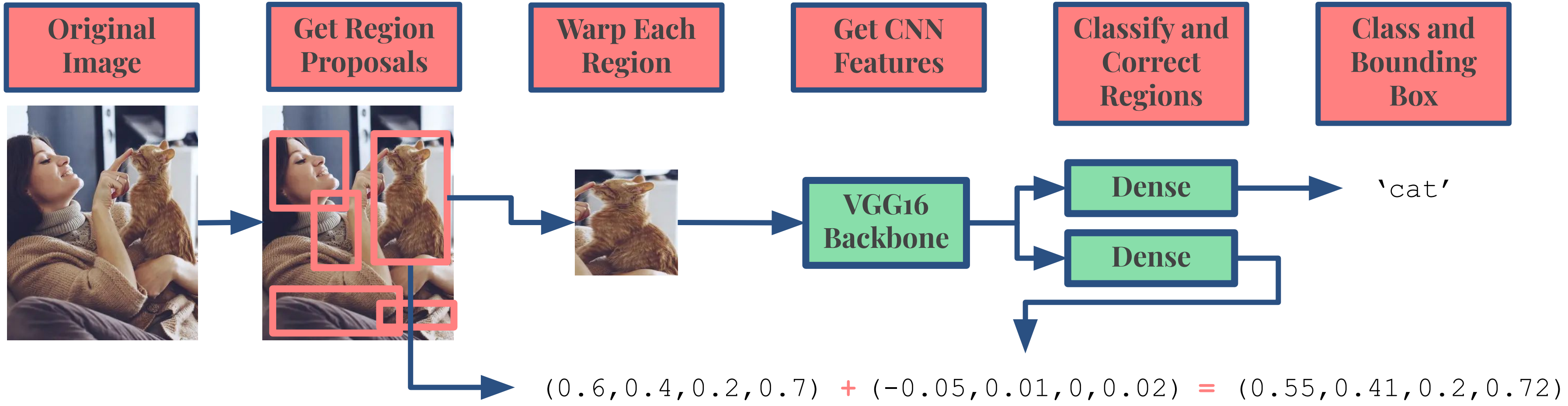
Improving Detection

- This method is, however, **inefficient for real data**.
- That is mainly due to the proposals not matching the objects they are looking for very well, producing an **offset** that makes detection imprecise.
- This offset is a vector of 4 dimensions of off the proposal's location is compared to the ground-truth's.
- In 2013, a team of researchers from UC Berkeley solved this problem by proposing the **Region-based Convolutional Neural Network (R-CNN)**.
- In R-CNN, the network not only predicts the class of each proposal, but also predicts the offset of that proposal with respect to the object on the image.



Object detection with R-CNN

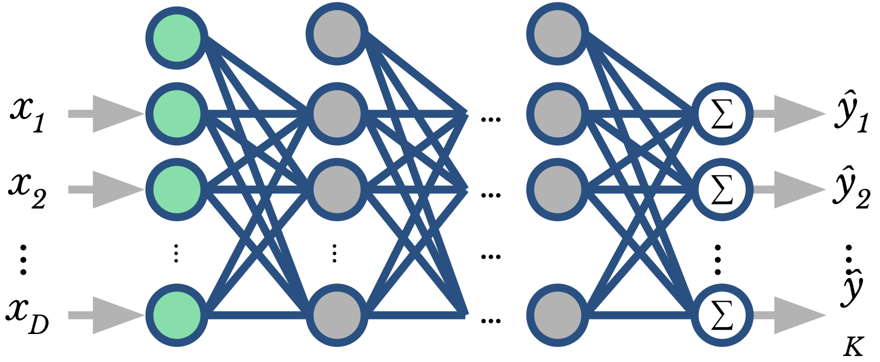
- The pipeline for R-CNN is similar to our previous approach, with the difference now that we training two MLPs (a sequence of dense layers) after the CNN block:
  - The first takes care of **classification**, like before.
  - The second performs **regression** on the offsets, i.e., how much we should shift a bounding boxe to align them better to the object.



Regression and MSE Loss

- In R-CNN, we are doing:
  - **Classification** for getting object classes.
  - **Regression** to find the bounding box offsets.
- In regression, as opposed to classification, the goal is to **predict a continuous value**, instead of a class.
- We implement a regressor (as opposed to a classifier) in an (dense) MLP by simply removing its softmax before the final output.
- The typical loss regression, the typical loss is **Mean Squared Error Loss (MSE)**, given by:

An MLP for Regression



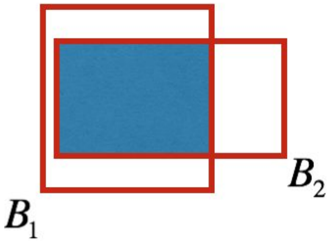
$$L(\theta) = \frac{1}{n}\sum_{i=1}^{n} l(\hat{y}^{(i)}, y^{(i)}), \text{ where } l(\hat{y}, y) = \|y - \hat{y}\|_2^2 = \sum_{j=1}^{K}(y_j - \hat{y}_j)^2$$

where $\{\hat{y}_1, \hat{y}_1, ..., \hat{y}_n\}$ are the predictions and $\{y_1, y_1, ..., y_n\}$ are the expected result.

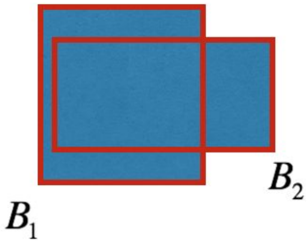
Measuring Performance

- Imagine a scenario where we came up with a prediction with R-CNN of a bounding box for an object. *How do we measure the accuracy of our prediction?*
- The concept of **Intersection over Union (IoU)** comes in handy in such a scenario:
  - Intersection measures how overlapping the predicted and actual bounding boxes are,
  - Union measures the overall space possible for overlap.
- IoU is the ratio of the overlapping region between the two bounding boxes over the combined region of both the bounding boxes and its value is always between 0 and 1.

Intersection

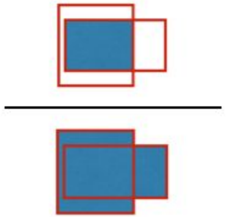


Union



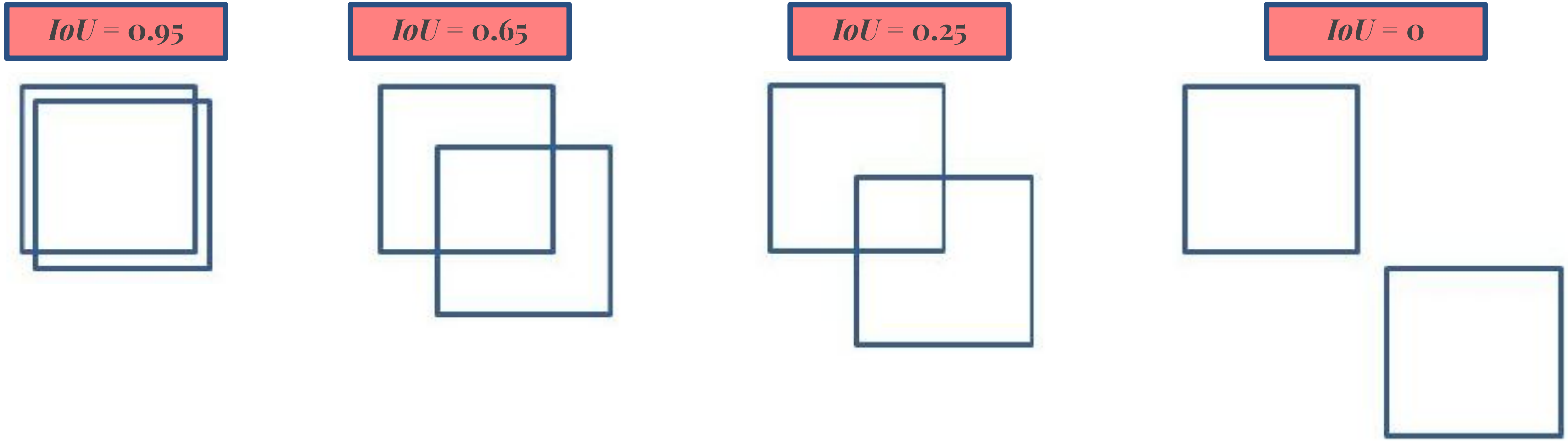
Intersection over Union

$$IoU = \frac{B_1 \cap B_2}{B_1 \cup B_2} =$$



Measuring Performance

- The larger the IoU, the greater the overlap between two regions, therefore the better the prediction compared to the ground truth bounding box.

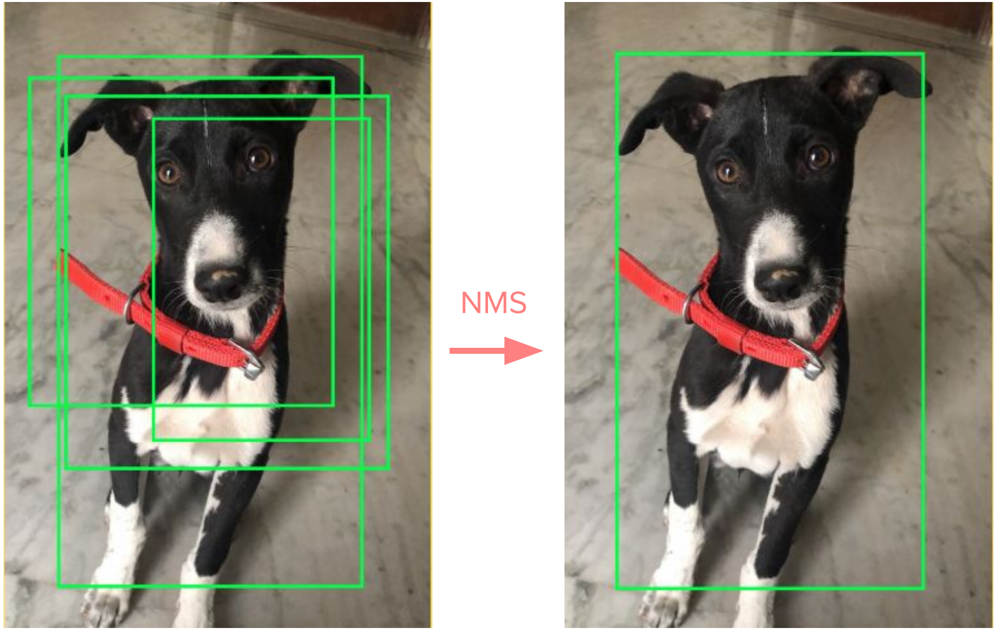


- In practice, we also set a threshold t to IoU such that if $IoU < t$, we say that the network didn't detect anything in that region (even if the class is correct), so it failed detection.

Non-maximum Suppression

- After we finish RCNN's inference, we may end up with many similar predictions on top of each other.
- Here, we use **Non-Maximum Suppression (NMS)** to solve this.
- In NMS, we try to suppress (i.e., delete) all predictions around an object that are not the maximum.
- PyTorch, we can use NMS via:

```
from torchvision.ops import nms
ixs = nms(bbs, confs, thr)
```



where `bbs` are the BBs and `confs` are the classification confidence of of each BB. It also discards all overlapping BBs with $IoU >$ `thr`.

Exercise (*In pairs*)

- Write and algorithm (not need to code here) that computes the IoU of two boxes using the `(Cx, Cy, H, W)` notation. Then write another algorithm for boxes that use the `(xmin, xmax, ymin, ymax)` notation.