[Tutorial] Object Detection

0. What we have seen so far : Image Classification

So far, we only know how to use CNN to do image classification: whether our image contains a dog or a cat.

This image is CC0 public domain

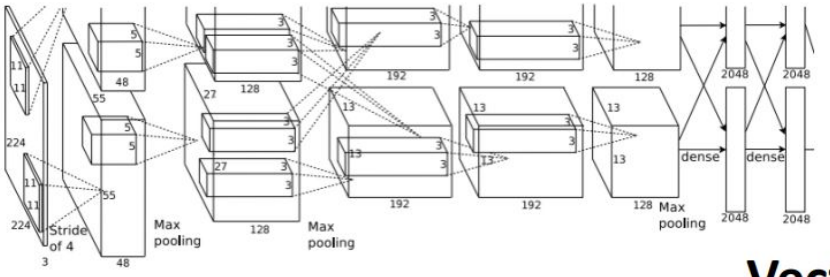
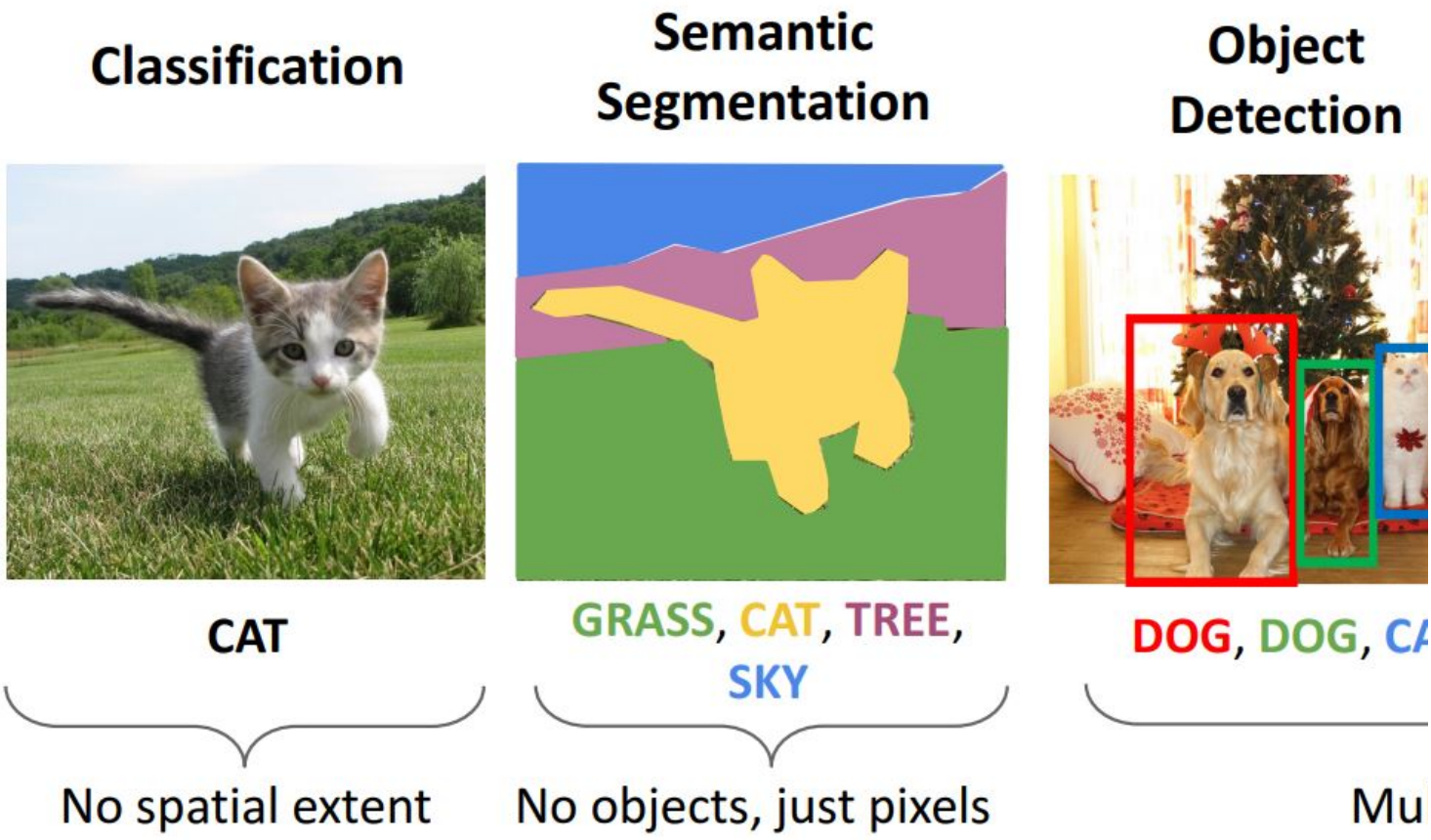


Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

0.1. Different Computer Vision Tasks



---

1. Object detection

1.1. Definition

It's all well and good to be able to classify images between cats and dogs ; but what if we wanted to know **where** in the image the cat is situated ?

➡️ That's where **object detection** comes into play.

In object detection :

- The **input** will be a single RGB image
- The **output** will be a set of detected object. The aim is to predict, for each object :
  - A **category label** (*What* is the object ?), selected from a fixed, known set of categories
  - A **bounding box** (*Where* is the object ?), usually made of four informations (x, y, width, heigth)

1.2. What is a Bounding Box ?

A way to get the position of an object in an image is to compute a **bounding box** around it. This is the most common way to get the position of an object in an image.

How to define a bounding box? With 4 values:

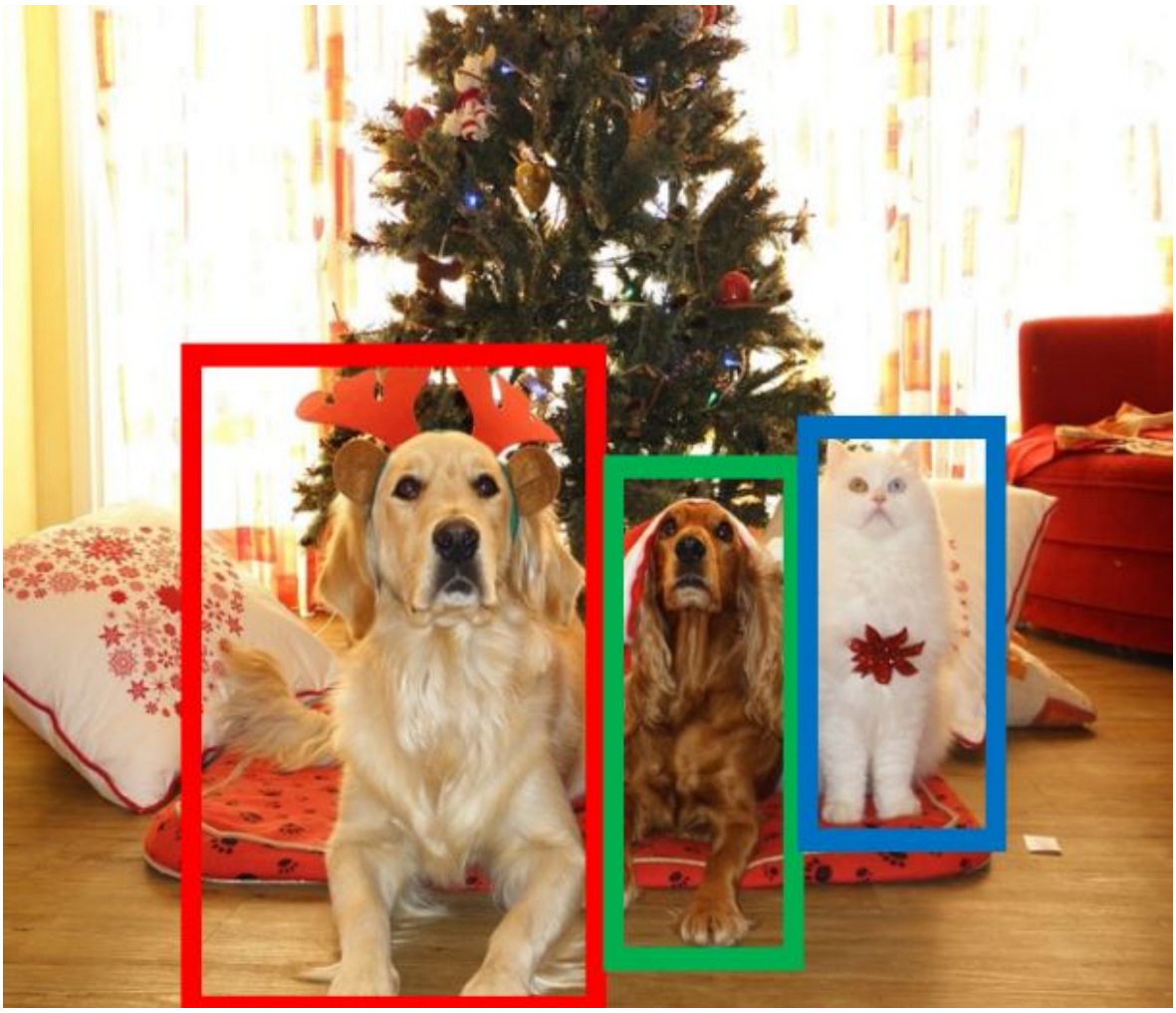
- The left top point b_x bx and b_y by of the bounding box
- The height b_h bh and width b_w bw of the bounding box

In our example picture with a car, the center of the bounding box would be (480, 400), for an height of 200 and a width of 300 pixels.

That would give that on our image with such a bounding box:

```
import matplotlib.pyplot as plt
import matplotlib.patches as patches

# Define the figure
fig, ax = plt.subplots(1, figsize=(12,8))

# Read the image
img = plt.imread('car_alone.png')

# Create a rectangle patch
rect = patches.Rectangle((480,400),300,200,linewidth=1,edgecolor='r',facecolor='none')

# Add the patch to the figure
ax.add_patch(rect)

# Display the image
plt.axis('off')
plt.imshow(img)
plt.show()
```

1.3. Why is it so hard ?

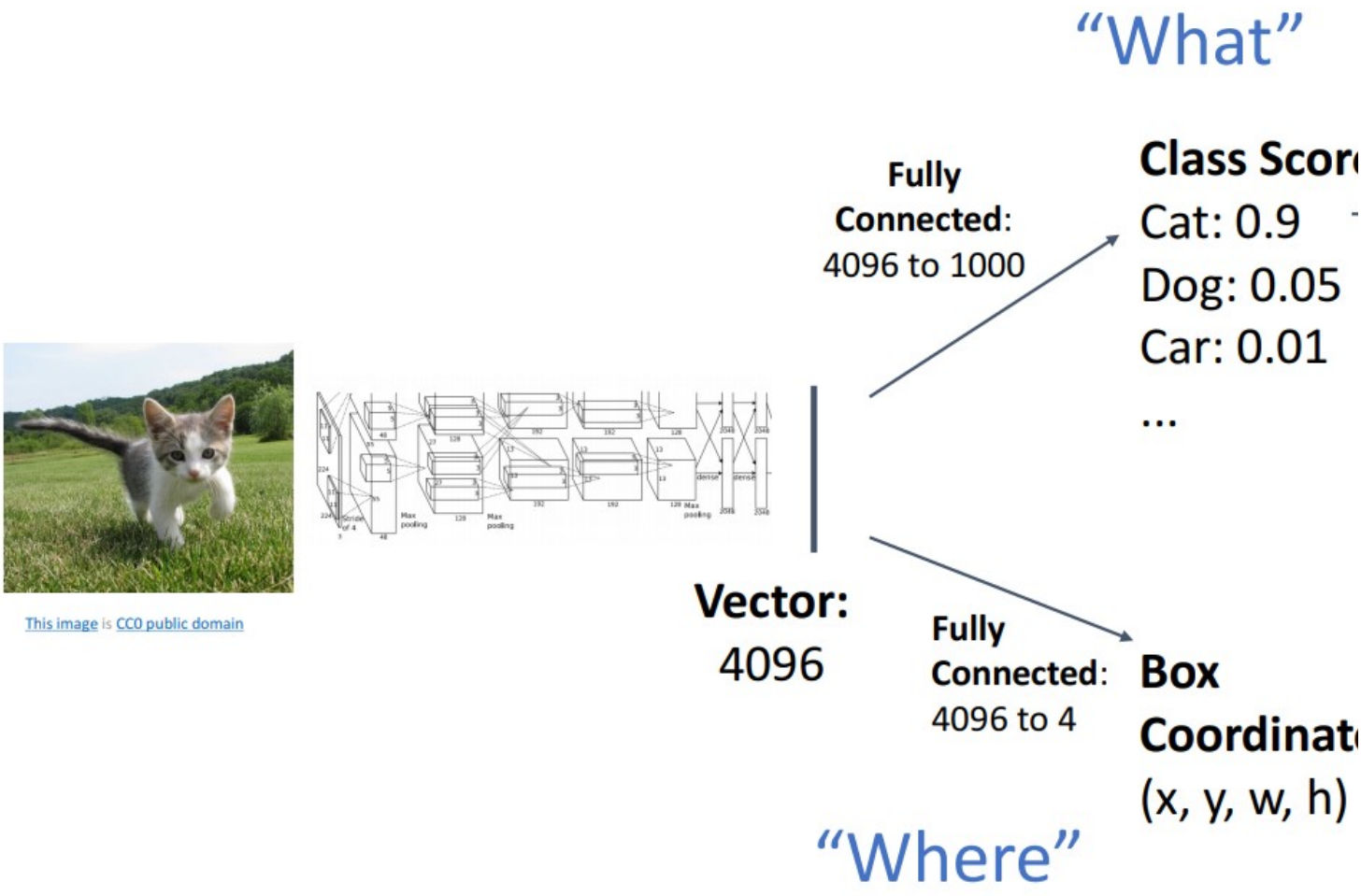


There are a number of factors that make the task of detecting objects, and their corresponding bounding boxes, quite complex :

- **Multiple ouputs** : Need to input variable numbers of objects per image
- **Multiple types of output**: Need to predict "what" (category label) as well as "where" (bounding box)
- **Large images**: Classification works at 224x224; need hight resolution for detection, often 800x600

---

2. Detecting a single object



1️⃣ **First step : Identify object : "What"**

- As seen so far, a simple classification is carried out, using a CNN.
- After the fully connected layer (you also know it under the name of Dense layer), we obtain an array of class scores, that indicate the probability of each class.
- With the correct label, we can apply a **softmax function** in order to calcule the loss.

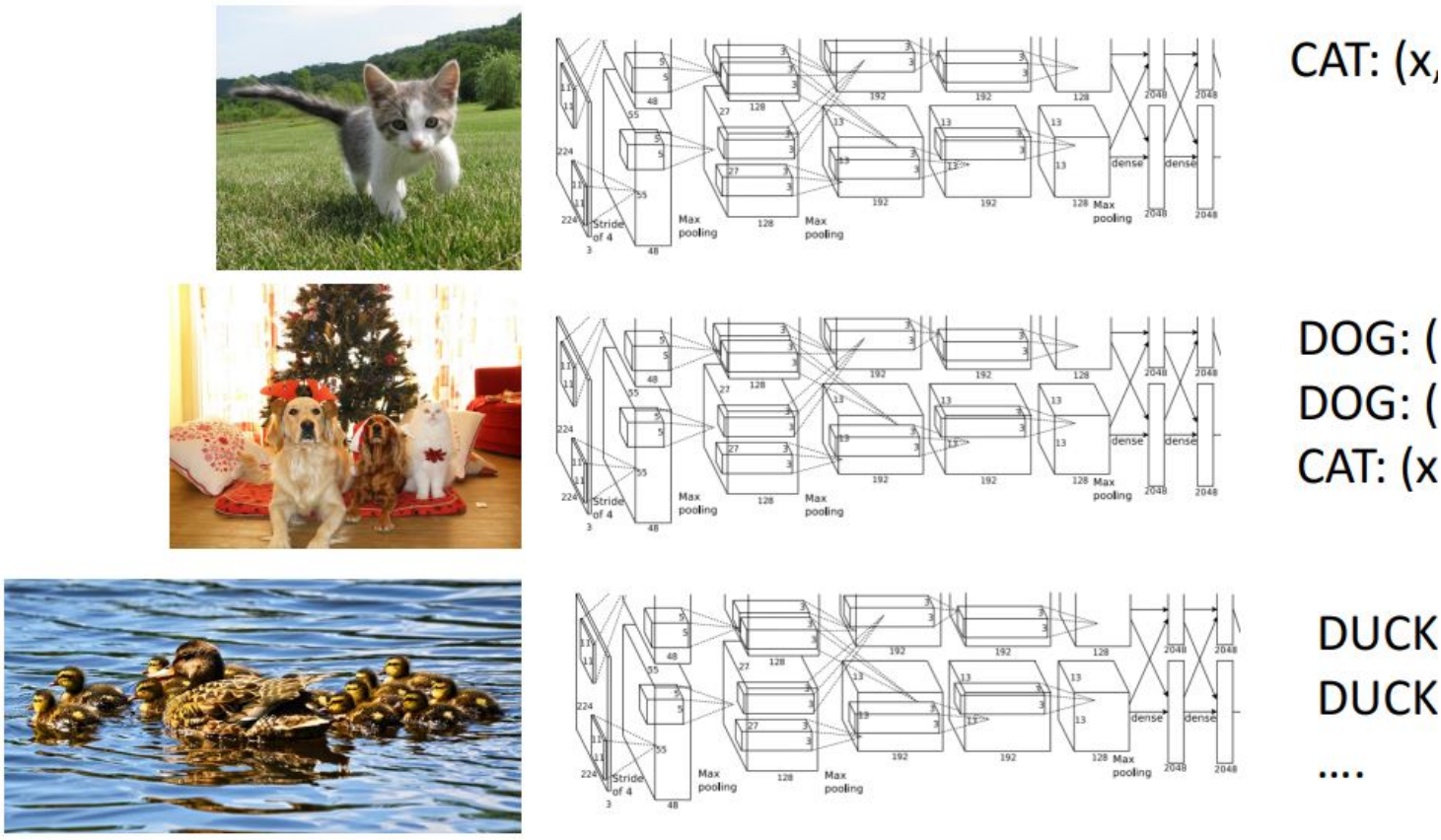
2️⃣ **Second Step : Localisation the object : "Where"**

- Here, we can just treat the localization as a regression problem : we need to predict four continuous variables (x, y, width, heigth of the bounding box)
- We usually use a **L2 loss** in this step.

3️⃣ **Combine Loss : Multitask loss**

- Now we have 2 differents loss : A softmax loss and a L2 loss ;
- **We can sum the two losses in order to have one multitask loss**
- Armed with this multitask loss, we can train our neural network to perform object detection !

⚠️ **But there is a problem : images can contain more than one object !**



---

3. Detecting Multiple Objects : Sliding Window

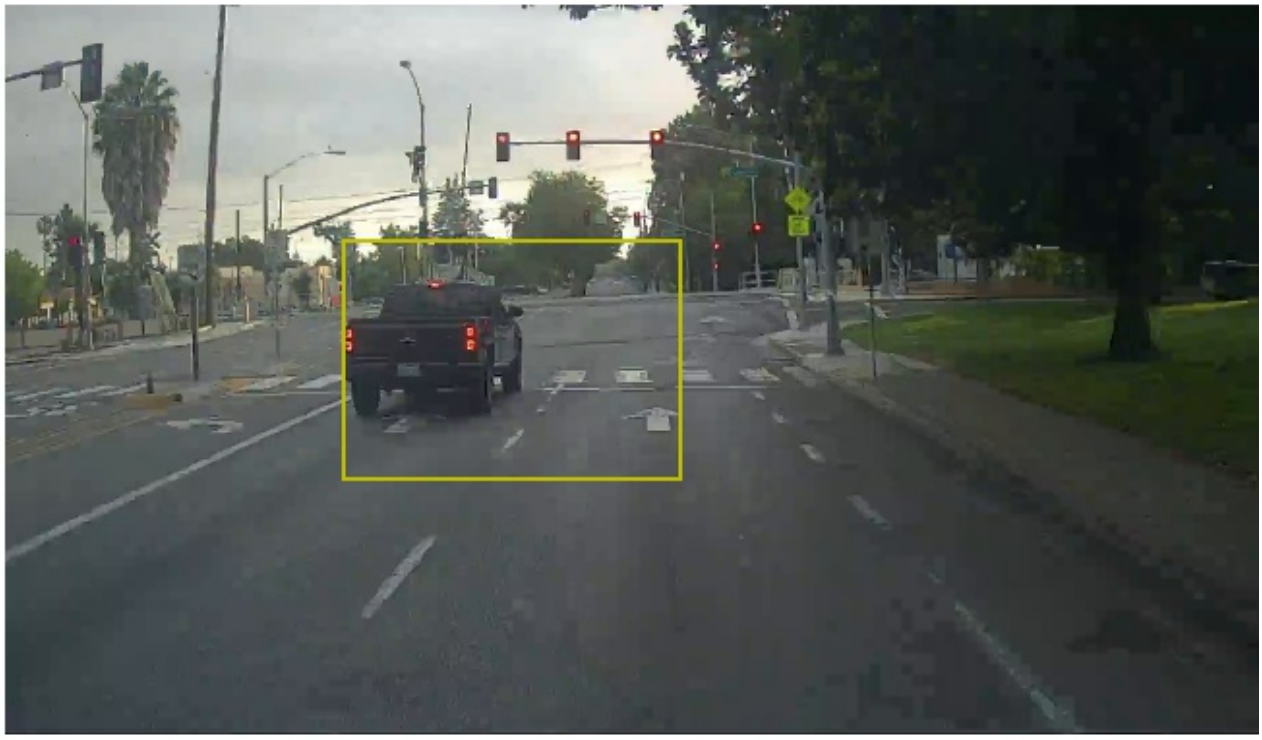
The idea :

- Let's apply a CNN to many different crops of an image ;
- Then the CNN classifies each crop as either object or background.

Using our previous image example :

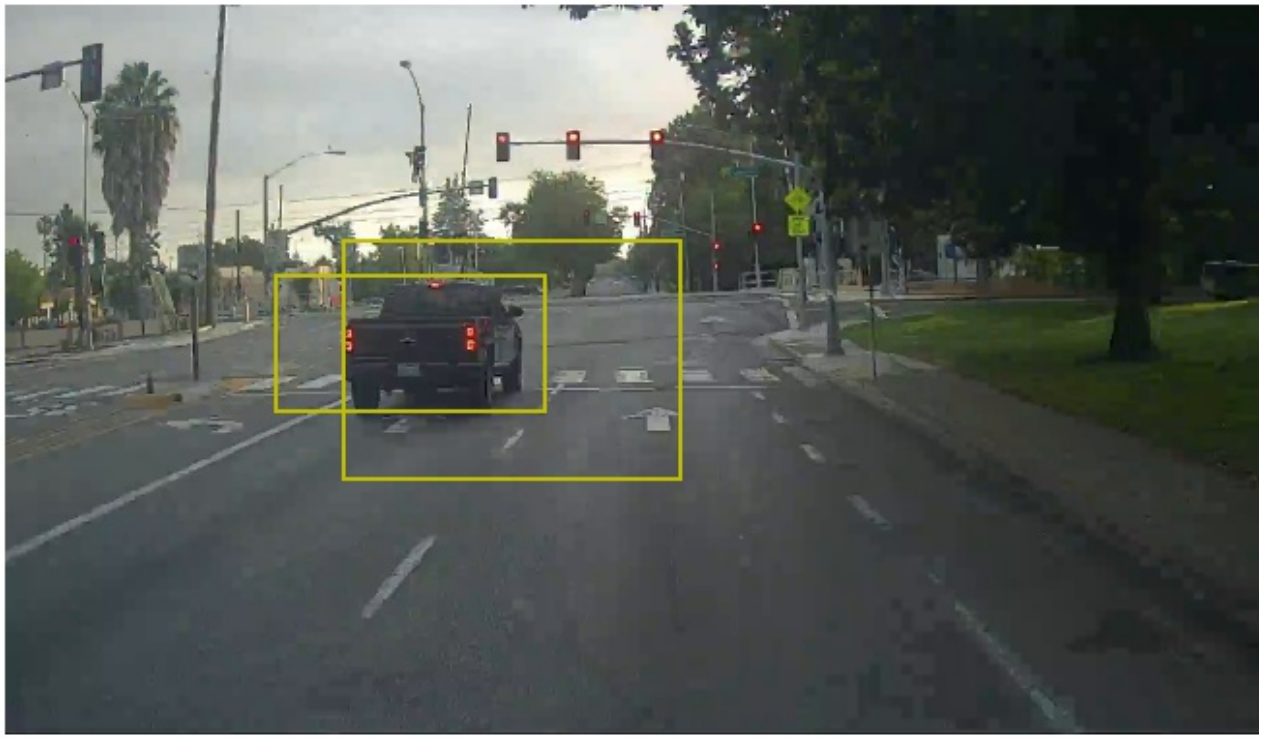
1. Use a sliding window to apply this binary classification on small areas of an image, to find where there is a car

Bingo! In one of our subimages, there is the car! But the result is not extremely precise, the box is way too big, which can be an issue.



1. Next step : using a smaller sliding window for a more precise detection.

So we finally end up with the following output:



Question: How many possible boxes are there in an image of size HxWHxW ?

Consider a box of size hxwhxw :

- Possible x positions: $W–w+1$W–w+1
- Possible y positions: $H–h+1$H–h+1
- Possible positions: $(W–w+1)*(H–h+1)$(W–w+1)∗(H–h+1)

😱 **A 800 x 600 image has ~58M boxes! No way we can evaluate them all**

This approcah is very computationally intensive : we have to loop over all the boxes in the image, for all the box sizes we want, and predict a CNN output for each iteration!

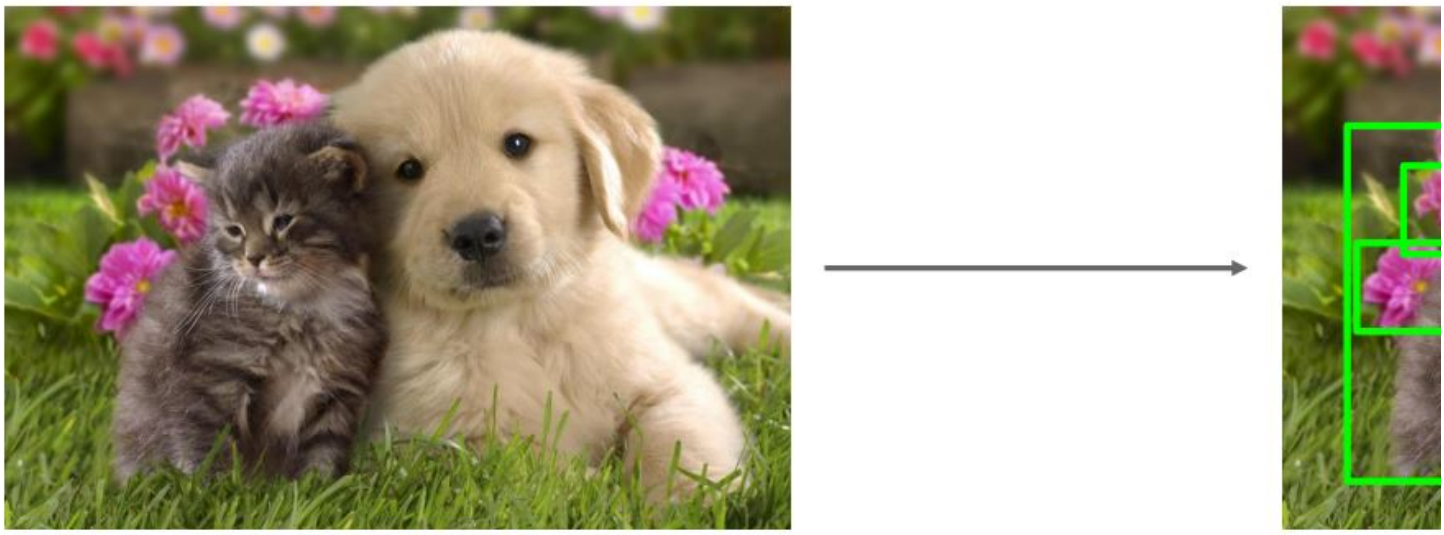
Let's see how to improve that!

---

4. A new concept : Region Proposals

The idea :

- Find a small set of boxes that are likely to cover all objects
- Often based on heuristics: e.g. look for “blob-like” image regions
- Relatively fast to run; e.g. Selective Search gives 2000 region proposals in a few seconds on CPU



> 📚 Some reading on the subject :
> **Alexe et al** (http://calvin-vision.net/wp-content/uploads/Publications/alexe12pami.pdf), “Measuring the objectness of image windows”, TPAMI 2012
> **Uijlings et al** (https://www.researchgate.net/publication/262270555_Selective_Search_for_Object_Recognition), “Selective Search for Object Recognition”, IJCV 2013
> **Cheng et al** (https://mmcheng.net/mftp/Papers/ObjectnessBING.pdf), “BING: Binarized normed gradients for objectness estimation at 300fps”, CVPR 2014
> **Zitnick and Dollar** (https://pdollar.github.io/files/papers/ZitnickDollarECCV14edgeBoxes.pdf), “Edge boxes: Locating object proposals from edges”, ECCV 2014

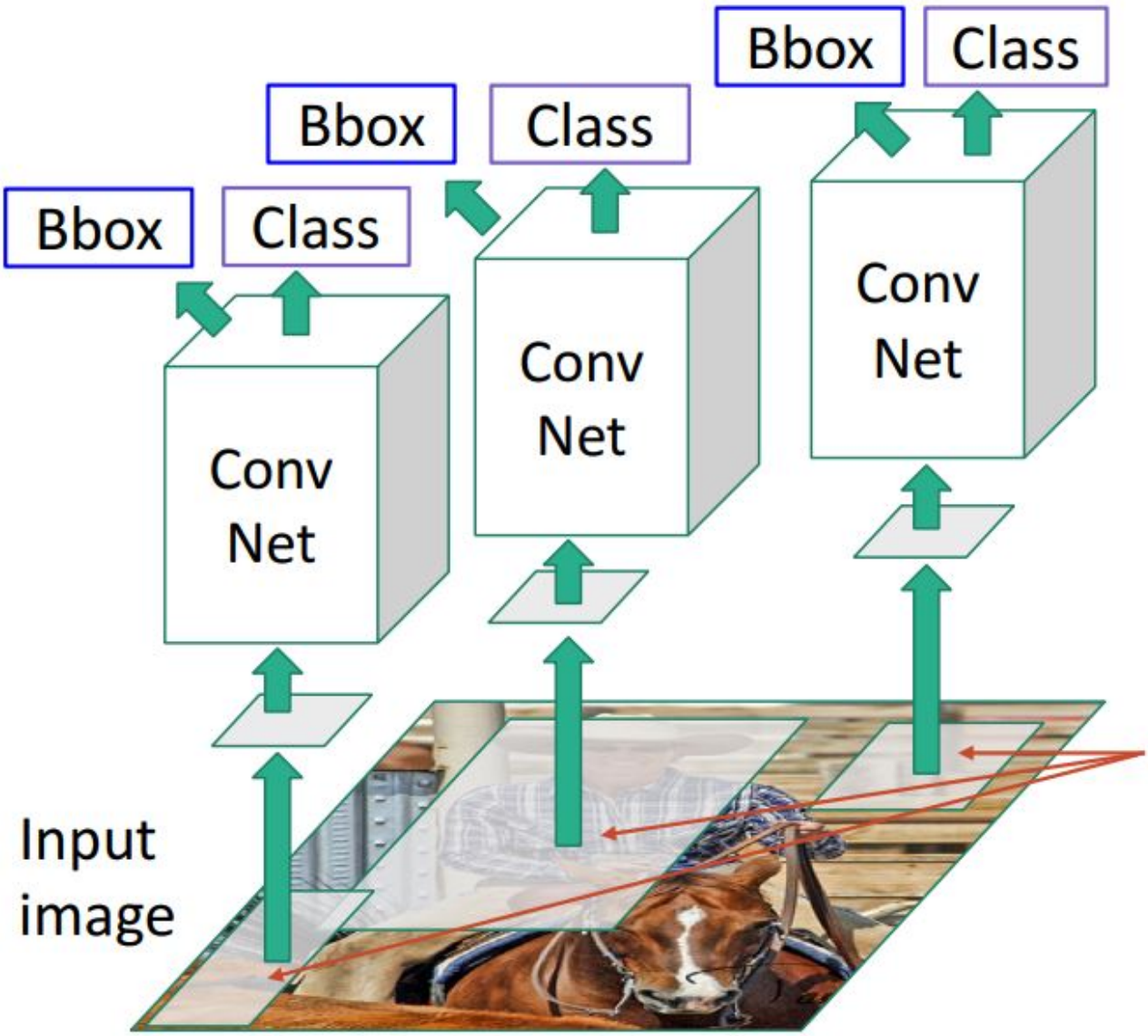
---

5. A first solution : R-CNN

The input: Single RGB Image

The process :

1. **Run region proposal method** to compute ~2000 region proposals
2. **Resize each region to 224x224 and run independently through a CNN to predict class scores and bounding boxes**
3. **Use scores to select a subset of region proposals to output** (Many choices here: threshold on background, or per-category? Or take top K proposals per image?)
4. **Compare with ground-truth boxes**



⚠️ One question remains : **How to compare boxes ?**

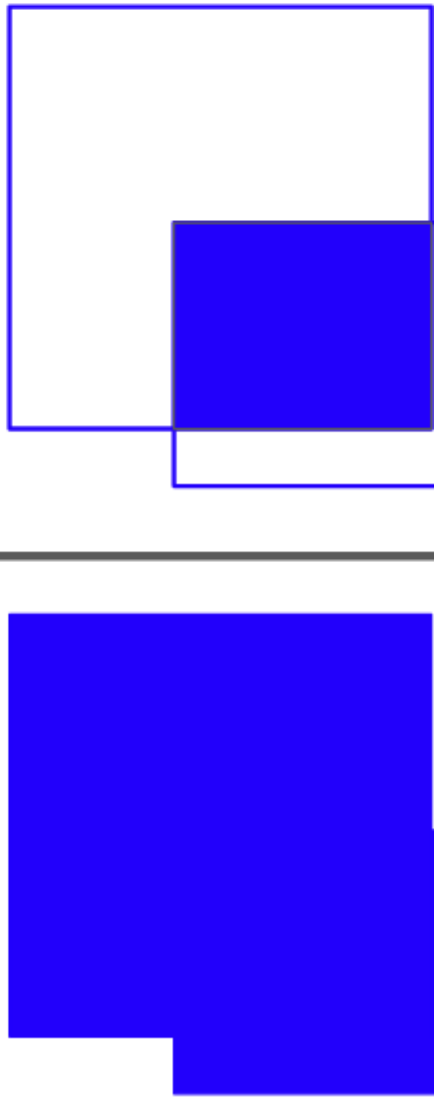
5.1. Comparing Boxes: Intersection over Union (IoU)

How can we compare our prediction to the ground-truth box?

We will use the **Intersection over Union (IoU)** (Also called “Jaccard similarity” or “Jaccard index”):

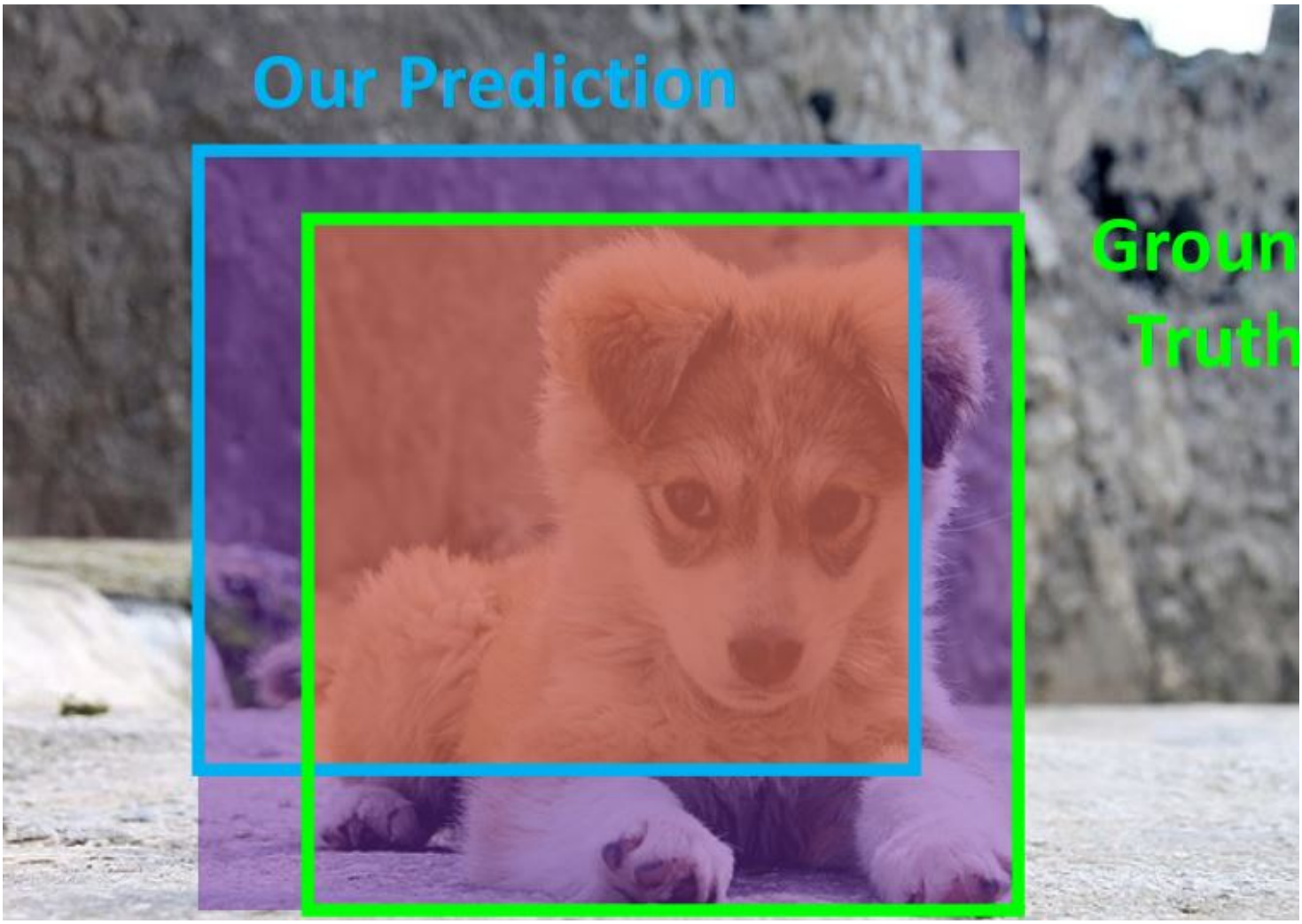
The IoU is defined as the intersection of two boxes over the union of the same boxes.

$$IoU = \frac{\text{Intersection}}{\text{Union}}$$



Quite intuitively, the IoU could be seen as a score ranging from 0 to 1:

- IoU=1: two boxes that perfectly overlap
- IoU=0: two boxes that do not overlap at all



5.2. Dealing with the problem of overlapping boxes : Non-max Suppression

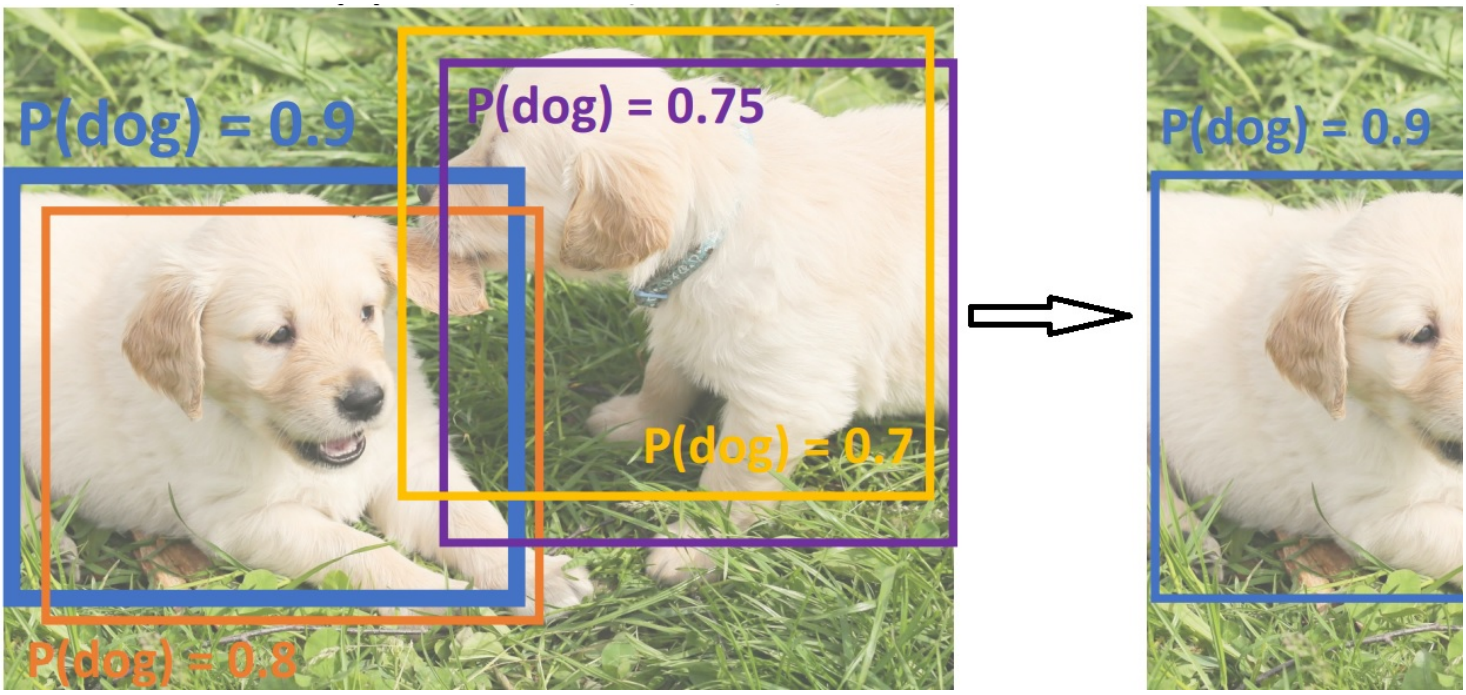
Problem: Object detectors often output many overlapping detections. If our model predicts more than one box for a given object, which one to keep?

➡️ **Solution**: Post-process raw detections using **Non-Max Suppression (NMS)**

The algorithm is the following:

1. Select the next highest-scoring box
2. Eliminate lower-scoring boxes with IoU > a given threshold (e.g. 0.7)
3. If any boxes remain, repeat steps 1 & 2

It would give this kind of results:



Problem: NMS may eliminate ”good” boxes when objects are highly overlapping… There is no perfect solution.

---

6. How to improve R-CNN ? Going fast and faster

R-CNN is good... But we still have a problem :

- **The network still takes a huge amount of time to train**, since you would have to classify 2000 region proposals per image.
- **The model cannot be easily implemented in real time**, since it takes around 47 seconds for each test image.
- The selective search algorithm is a fixed algorithm. Therefore, no learning is happening at that stage. This could lead to the **generation of bad candidate region proposals**.

➡️ How can we improve R-CNN ? Can we apply on forward passes instead of 2k ?

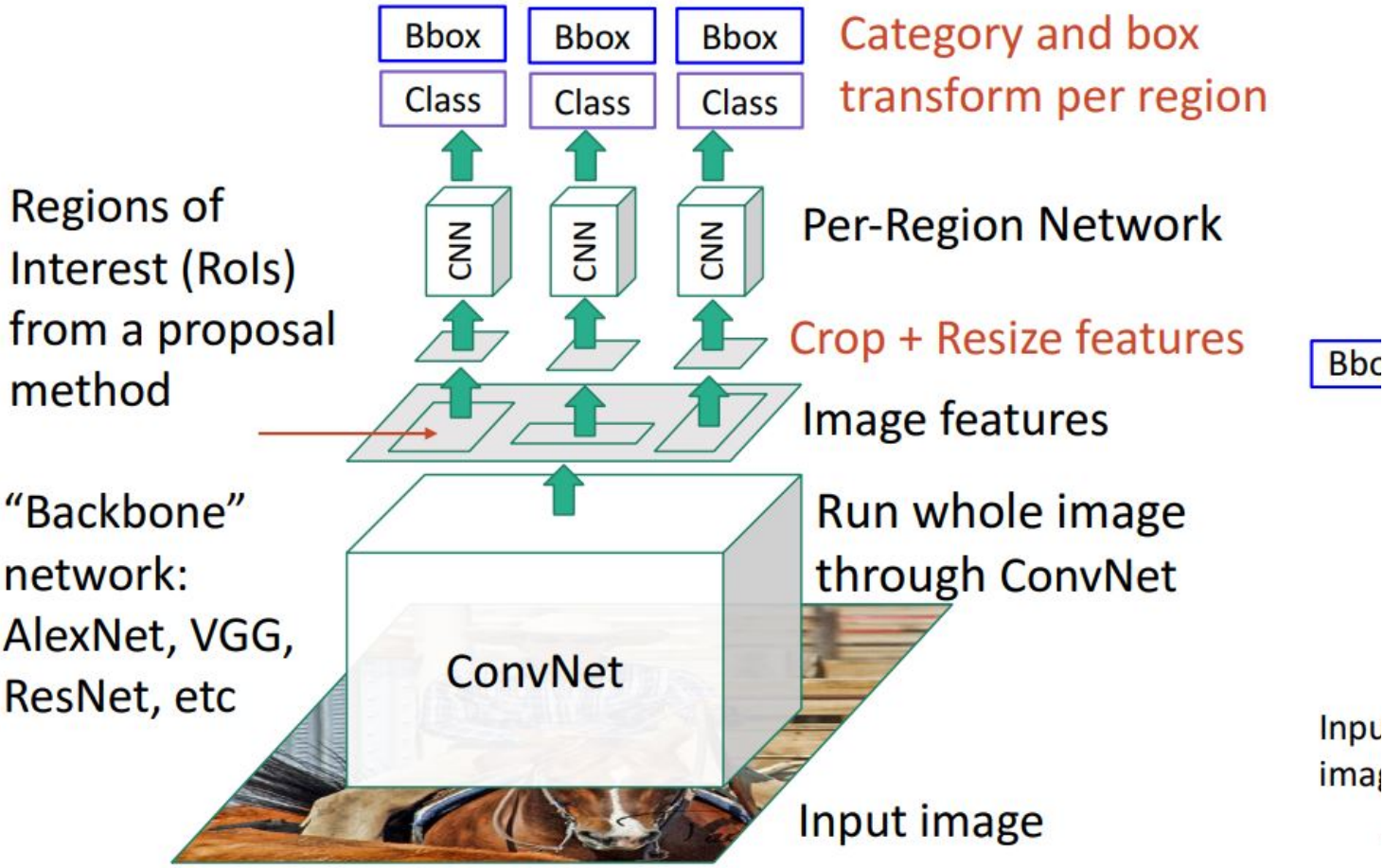
6.1. Fast-RCNN

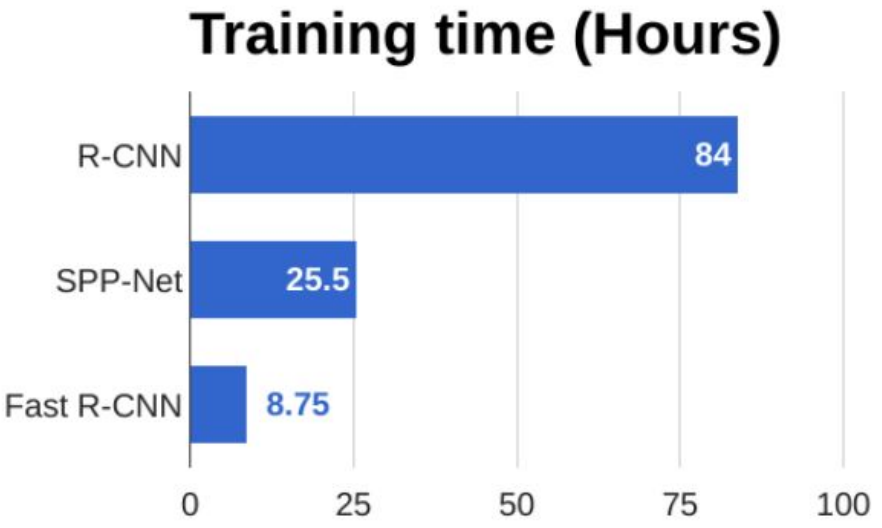
The same author of the previous paper (R-CNN) solved some of the drawbacks of R-CNN to build a faster object detection algorithm : the Fast R-CNN. The approach is similar to the R-CNN algorithm.

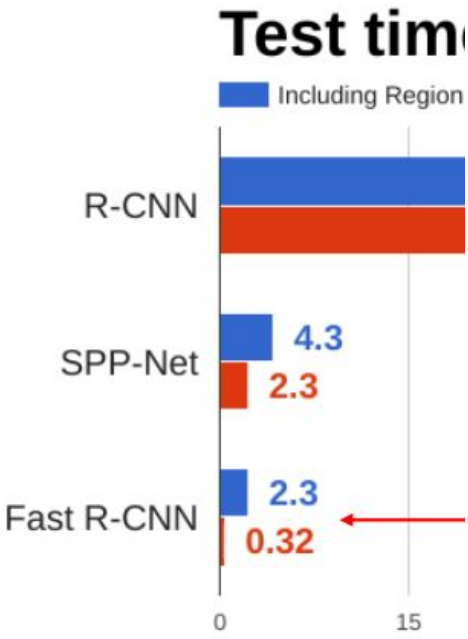
But, instead of feeding the region proposals to the CNN, we feed :

- The input image is fed into the CNN to **generate a convolutional feature map**.
- Then from the convolutional feature map, we **identify the region of proposals** and warp them into squares ; by using a RoI pooling layer, we reshape them into a fixed size so that it can be fed **into a fully connected layer**.

- From the RoI feature vector, we use a **softmax layer** to **predict the class of the proposed region** and also **the offset values for the bounding boxes**.







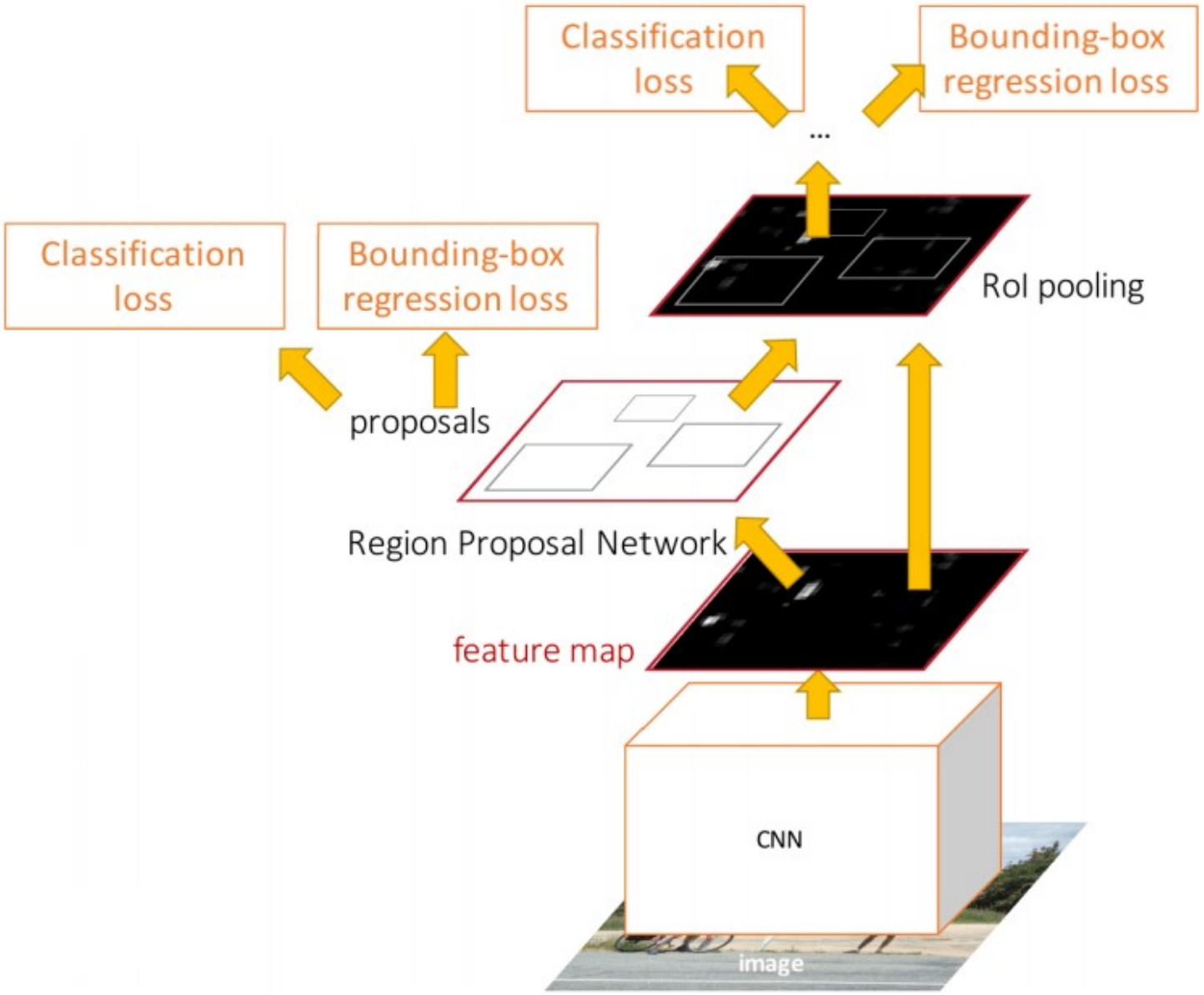
- Fast R-CNN is significantly faster in training and testing sessions over R-CNN

- Including region proposals slows down the algorithm significantly when compared to not using region proposals. Therefore, **region proposals become bottlenecks in Fast R-CNN algorithm**, affecting its performance.

6.2. Faster-RCNN

The idea of the **FAster-RCNN** is to **eliminate the selective search algorithm** and **let the network learn the region proposals**.

- Similar to Fast R-CNN, the image is provided as an input to a convolutional network which provides a convolutional feature map.
- Instead of using selective search algorithm on the feature map to identify the region proposals, a separate network is used to predict the region proposals ➡️ **Region Proposal Network**



---

7. Single-Stage Object Detection with YOLO

7.1. So far : two-stage objct detection

So far, Faster R-CNN is **two-stage object detector**.

1️⃣ **First Stage** : Run once per image

- Backbone network
- Region proposal network

2️⃣ **Second Stage** : Run once per region

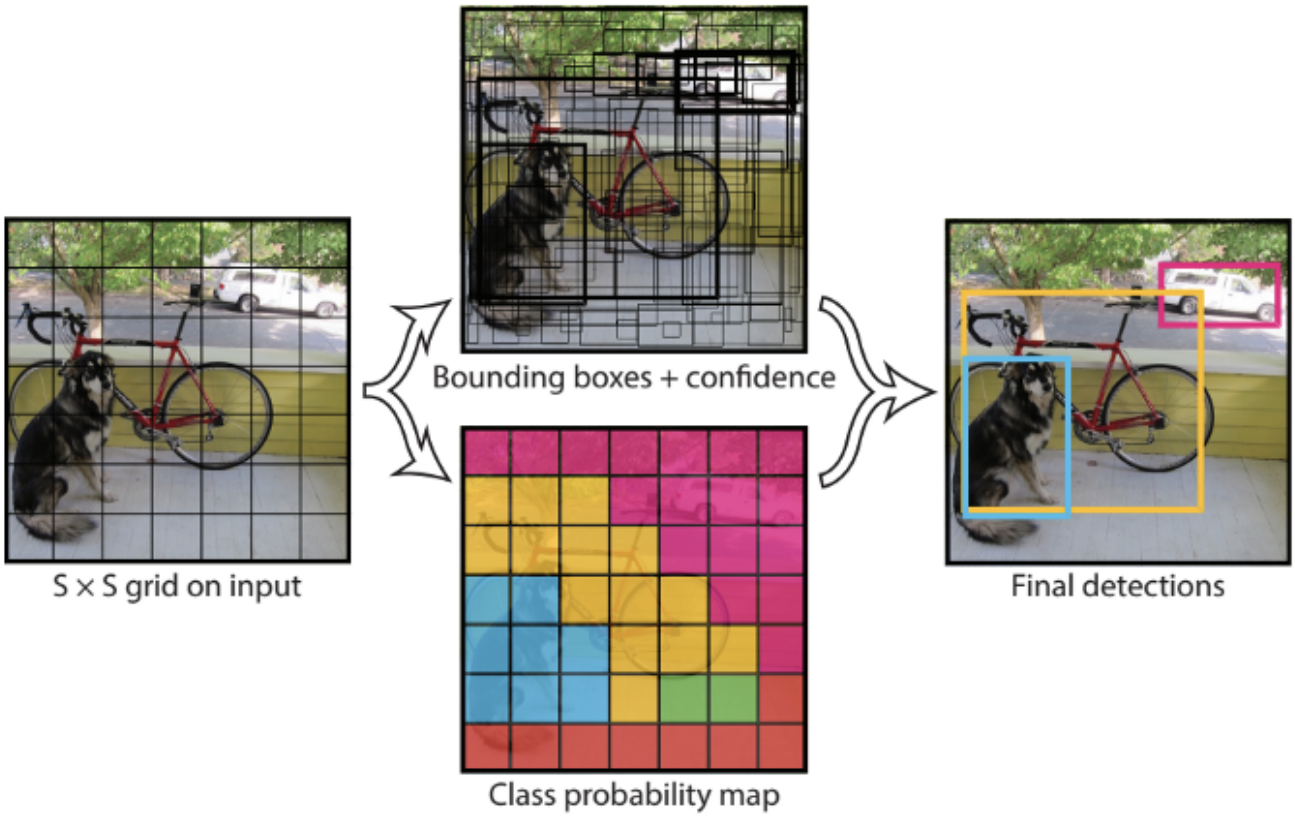
- Crop features : RoI Pool
- Predict objects classes
- Prediction bbox offset

🤔 **But do we really need the second stage ?**

7.2. Yolo : You Only look Once !

All of the previous object detection algorithms use regions to localize the object within the image. Thus, the network never looks at the complete image.

For this reason, YOLO is vastly different from the region based algorithms: it uses a **single convolutional network that predicts the bounding boxes and the class probabilities for these boxes**.

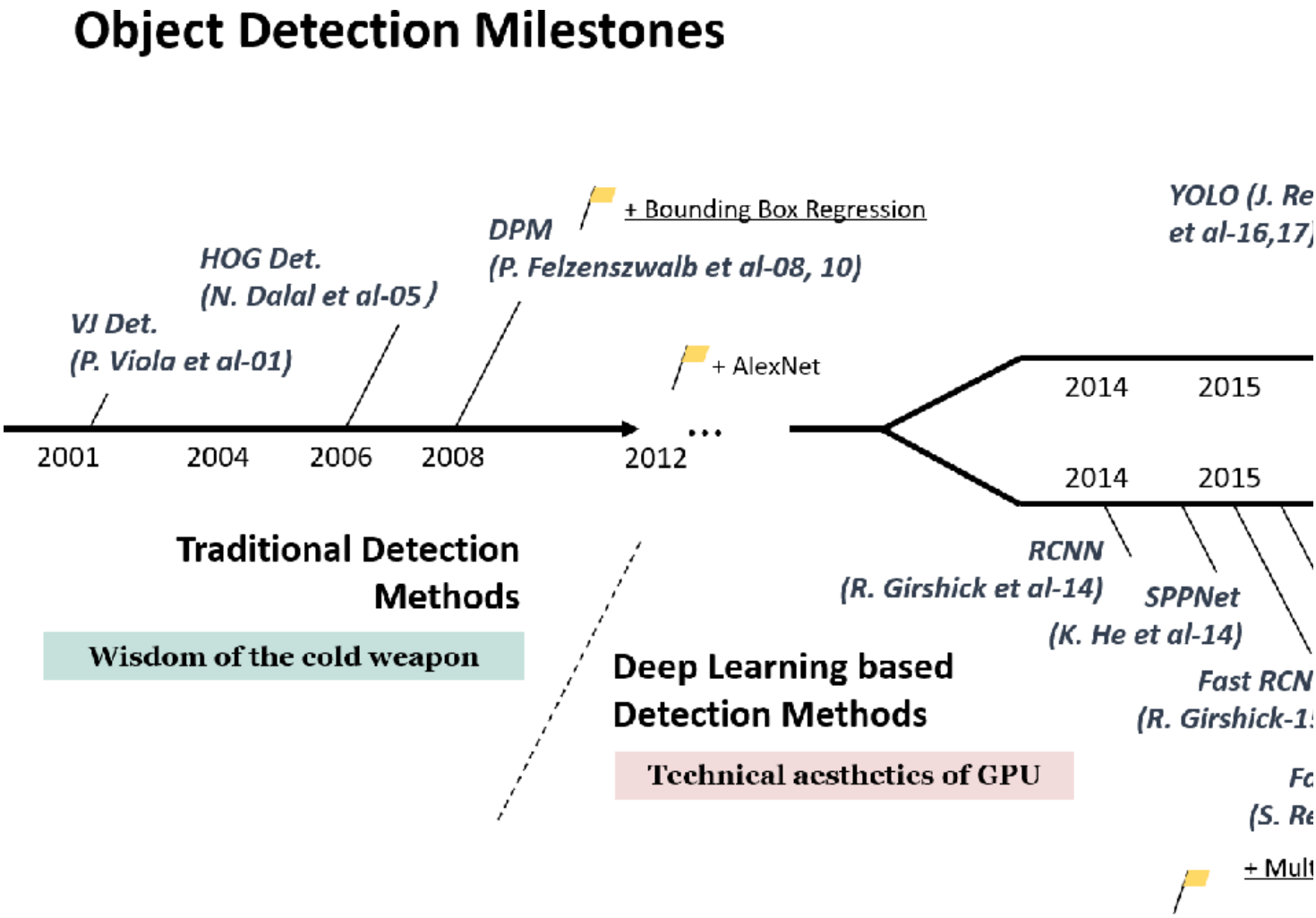


How does it work ?

- First, YOLO takes an image and splits it into an SxS grid
- Each cell (or grid) predicts a Bouding Box
- And returns the bouding boxes above a given confidence threshold.
- **YOLO is vastly faster** (45 frames per second) than any other object detection algorithm.
- **Limitation** : It struggles with small objects within the image.

---

8. The history of Object Detection



Comments

- Until 2017 :
  - Two Stage method (faster R-CNN) get the best accuracy, but are slower
  - Single-Stage methods (SSD, Yolo) are much faster, but don't perform as well
  - Bigger backbones improve performance, but are slower
- **But** : Since then, GPUs have gotten faster and performances are improved with many tricks :
  - Train longer

- MultiScale Backbone : Feature Pyramid Network
- Better backbone : ResNeXt, EfficientNet
- More data, big models work better

➡️ Single-Stage methods have improved and now achieve state of the art !

---

9. Evaluating Object Detectors: Mean Average Precision (mAP)

To evaluate object detection models like R-CNN and YOLO, the **mean average precision (mAP)** is used. **The mAP compares the ground-truth bounding box to the detected box and returns a score**. The higher the score, the more accurate the model is in its detections.

How to compute the mAP ?

1. **Run object detector on all test images (with NMS)**
2. **For each category, compute Average Precision (AP)** = area under Precision vs Recall Curve
    1. For each detection (highest score to lowest score) :
       a.If it matches some GT box with IoU > 0.5, mark it as positive and eliminate the GT
       b. Otherwise mark it as negative
       c. Plot a point on PR Curve
    2. Average Precision (AP) = area under PR curve
3. **Mean Average Precision (mAP) = average of AP for each category**
4. **For "COCO mAP": Compute mAP@thresh for each IoU threshold** (0.5, 0.55, 0.6, …, 0.95) **and take average**

---

10. Implementation : Object Detection with Detectron2

Object detection is hard! Don't implement it yourself

Detectron2 is Facebook's new library that implements **state-of-the-art object detection algorithms**. We will see how to use Detectron2 for both inference as well as using transfer learning to train on your own dataset.

How to Use it

⚠️ **Run the following code on Google colab, with the GPU enabled, not on your local notebook.**

Install detectron2

```
!pip install pyyaml==5.1
!pip install --upgrade torch
import torch, torchvision
print(torch.__version__, torch.cuda.is_available())
!gcc --version
```

```
import torch
assert torch.__version__.startswith("1.7")
!pip install detectron2 -f https://dl.fbaipublicfiles.com/detectron2/wheels/cu101/torch1.7/index.html
```

import package

```
from detectron2 import model_zoo
from detectron2.engine import DefaultPredictor
from detectron2.config import get_cfg
from detectron2.utils.visualizer import Visualizer
from detectron2.data import MetadataCatalog
import matplotlib.pyplot as plt
import cv2
```

```
!wget http://images.cocodataset.org/val2017/000000439715.jpg -O input.jpg
im = cv2.imread("./input.jpg")
```

Now, you have to define configuration in order to load model.

- for this, `get_cfg()` will define the default configuration
- then, you have to chose the backbone and architecture you want to use and specify it in config

In this example, we will use a **faster_rcnn** with **resnet** backbone :

```
cfg = get_cfg()
cfg.merge_from_file(model_zoo.get_config_file("COCO-Detection/faster_rcnn_R_101_FPN_3x.yaml"))
cfg.MODEL.ROI_HEADS.SCORE_THRESH_TEST = 0.5  # set threshold for this model
cfg.MODEL.WEIGHTS = model_zoo.get_checkpoint_url("COCO-Detection/faster_rcnn_R_101_FPN_3x.yaml")
```

Print the whole configuration

```
print(cfg)
```

```
CUDNN_BENCHMARK: False
DATALOADER:
  ASPECT_RATIO_GROUPING: True
  FILTER_EMPTY_ANNOTATIONS: True
  NUM_WORKERS: 4
  REPEAT_THRESHOLD: 0.0
  SAMPLER_TRAIN: TrainingSampler
DATASETS:
  PRECOMPUTED_PROPOSAL_TOPK_TEST: 1000
  PRECOMPUTED_PROPOSAL_TOPK_TRAIN: 2000
  PROPOSAL_FILES_TEST: ()
  PROPOSAL_FILES_TRAIN: ()
  TEST: ('coco_2017_val',)
  TRAIN: ('coco_2017_train',)
GLOBAL:
  HACK: 1.0
INPUT:
  CROP:
    ENABLED: False
    SIZE: [0.9, 0.9]
    TYPE: relative_range
  FORMAT: BGR
  MASK_FORMAT: polygon
  MAX_SIZE_TEST: 1333
  MAX_SIZE_TRAIN: 1333
  MIN_SIZE_TEST: 800
  MIN_SIZE_TRAIN: (640, 672, 704, 736, 768, 800)
  MIN_SIZE_TRAIN_SAMPLING: choice
  RANDOM_FLIP: horizontal
MODEL:
  ANCHOR_GENERATOR:
    ANGLES: [[-90, 0, 90]]
    ASPECT_RATIOS: [[0.5, 1.0, 2.0]]
    NAME: DefaultAnchorGenerator
    OFFSET: 0.0
    SIZES: [[32], [64], [128], [256], [512]]
  BACKBONE:
    FREEZE_AT: 2
    NAME: build_resnet_fpn_backbone
  DEVICE: cuda
  FPN:
    FUSE_TYPE: sum
    IN_FEATURES: ['res2', 'res3', 'res4', 'res5']
    NORM:
    OUT_CHANNELS: 256
  KEYPOINT_ON: False
  LOAD_PROPOSALS: False
  MASK_ON: False
  META_ARCHITECTURE: GeneralizedRCNN
  PANOPTIC_FPN:
    COMBINE:
      ENABLED: True
      INSTANCES_CONFIDENCE_THRESH: 0.5
      OVERLAP_THRESH: 0.5
      STUFF_AREA_LIMIT: 4096
```

```
  INSTANCE_LOSS_WEIGHT: 1.0
PIXEL_MEAN: [103.53, 116.28, 123.675]
PIXEL_STD: [1.0, 1.0, 1.0]
PROPOSAL_GENERATOR:
  MIN_SIZE: 0
  NAME: RPN
RESNETS:
  DEFORM_MODULATED: False
  DEFORM_NUM_GROUPS: 1
  DEFORM_ON_PER_STAGE: [False, False, False, False]
  DEPTH: 101
  NORM: FrozenBN
  NUM_GROUPS: 1
  OUT_FEATURES: ['res2', 'res3', 'res4', 'res5']
  RES2_OUT_CHANNELS: 256
  RES5_DILATION: 1
  STEM_OUT_CHANNELS: 64
  STRIDE_IN_1X1: True
  WIDTH_PER_GROUP: 64
RETINANET:
  BBOX_REG_LOSS_TYPE: smooth_l1
  BBOX_REG_WEIGHTS: (1.0, 1.0, 1.0, 1.0)
  FOCAL_LOSS_ALPHA: 0.25
  FOCAL_LOSS_GAMMA: 2.0
  IN_FEATURES: ['p3', 'p4', 'p5', 'p6', 'p7']
  IOU_LABELS: [0, -1, 1]
  IOU_THRESHOLDS: [0.4, 0.5]
  NMS_THRESH_TEST: 0.5
  NORM:
  NUM_CLASSES: 80
  NUM_CONVS: 4
  PRIOR_PROB: 0.01
  SCORE_THRESH_TEST: 0.05
  SMOOTH_L1_LOSS_BETA: 0.1
  TOPK_CANDIDATES_TEST: 1000
ROI_BOX_CASCADE_HEAD:
  BBOX_REG_WEIGHTS: ((10.0, 10.0, 5.0, 5.0), (20.0, 20.0, 10.0, 10.0), (30.0, 30.0, 15.0, 15.0))
  IOUS: (0.5, 0.6, 0.7)
ROI_BOX_HEAD:
  BBOX_REG_LOSS_TYPE: smooth_l1
  BBOX_REG_LOSS_WEIGHT: 1.0
  BBOX_REG_WEIGHTS: (10.0, 10.0, 5.0, 5.0)
  CLS_AGNOSTIC_BBOX_REG: False
  CONV_DIM: 256
  FC_DIM: 1024
  NAME: FastRCNNConvFCHead
  NORM:
  NUM_CONV: 0
  NUM_FC: 2
  POOLER_RESOLUTION: 7
  POOLER_SAMPLING_RATIO: 0
  POOLER_TYPE: ROIAlignV2
  SMOOTH_L1_BETA: 0.0
  TRAIN_ON_PRED_BOXES: False
ROI_HEADS:
  BATCH_SIZE_PER_IMAGE: 512
```

```
  IN_FEATURES: ['p2', 'p3', 'p4', 'p5']
  IOU_LABELS: [0, 1]
  IOU_THRESHOLDS: [0.5]
  NAME: StandardROIHeads
  NMS_THRESH_TEST: 0.5
  NUM_CLASSES: 80
  POSITIVE_FRACTION: 0.25
  PROPOSAL_APPEND_GT: True
  SCORE_THRESH_TEST: 0.5
ROI_KEYPOINT_HEAD:
  CONV_DIMS: (512, 512, 512, 512, 512, 512, 512, 512)
  LOSS_WEIGHT: 1.0
  MIN_KEYPOINTS_PER_IMAGE: 1
  NAME: KRCNNConvDeconvUpsampleHead
  NORMALIZE_LOSS_BY_VISIBLE_KEYPOINTS: True
  NUM_KEYPOINTS: 17
  POOLER_RESOLUTION: 14
  POOLER_SAMPLING_RATIO: 0
  POOLER_TYPE: ROIAlignV2
ROI_MASK_HEAD:
  CLS_AGNOSTIC_MASK: False
  CONV_DIM: 256
  NAME: MaskRCNNConvUpsampleHead
  NORM:
  NUM_CONV: 4
  POOLER_RESOLUTION: 14
  POOLER_SAMPLING_RATIO: 0
  POOLER_TYPE: ROIAlignV2
RPN:
  BATCH_SIZE_PER_IMAGE: 256
  BBOX_REG_LOSS_TYPE: smooth_l1
  BBOX_REG_LOSS_WEIGHT: 1.0
  BBOX_REG_WEIGHTS: (1.0, 1.0, 1.0, 1.0)
  BOUNDARY_THRESH: -1
  HEAD_NAME: StandardRPNHead
  IN_FEATURES: ['p2', 'p3', 'p4', 'p5', 'p6']
  IOU_LABELS: [0, -1, 1]
  IOU_THRESHOLDS: [0.3, 0.7]
  LOSS_WEIGHT: 1.0
  NMS_THRESH: 0.7
  POSITIVE_FRACTION: 0.5
  POST_NMS_TOPK_TEST: 1000
  POST_NMS_TOPK_TRAIN: 1000
  PRE_NMS_TOPK_TEST: 1000
  PRE_NMS_TOPK_TRAIN: 2000
  SMOOTH_L1_BETA: 0.0
SEM_SEG_HEAD:
  COMMON_STRIDE: 4
  CONVS_DIM: 128
  IGNORE_VALUE: 255
  IN_FEATURES: ['p2', 'p3', 'p4', 'p5']
  LOSS_WEIGHT: 1.0
  NAME: SemSegFPNHead
  NORM: GN
  NUM_CLASSES: 54
WEIGHTS: https://dl.fbaipublicfiles.com/detectron2/COCO-Detection/faster_rcnn_R_101_FPN_3x/1378512
```

```
57/model_final_f6e8b1.pkl
OUTPUT_DIR: ./output
SEED: -1
SOLVER:
  AMP:
    ENABLED: False
  BASE_LR: 0.02
  BIAS_LR_FACTOR: 1.0
  CHECKPOINT_PERIOD: 5000
  CLIP_GRADIENTS:
    CLIP_TYPE: value
    CLIP_VALUE: 1.0
    ENABLED: False
    NORM_TYPE: 2.0
  GAMMA: 0.1
  IMS_PER_BATCH: 16
  LR_SCHEDULER_NAME: WarmupMultiStepLR
  MAX_ITER: 270000
  MOMENTUM: 0.9
  NESTEROV: False
  REFERENCE_WORLD_SIZE: 0
  STEPS: (210000, 250000)
  WARMUP_FACTOR: 0.001
  WARMUP_ITERS: 1000
  WARMUP_METHOD: linear
  WEIGHT_DECAY: 0.0001
  WEIGHT_DECAY_BIAS: 0.0001
  WEIGHT_DECAY_NORM: 0.0
TEST:
  AUG:
    ENABLED: False
    FLIP: True
    MAX_SIZE: 4000
    MIN_SIZES: (400, 500, 600, 700, 800, 900, 1000, 1100, 1200)
  DETECTIONS_PER_IMAGE: 100
  EVAL_PERIOD: 0
  EXPECTED_RESULTS: []
  KEYPOINT_OKS_SIGMAS: []
  PRECISE_BN:
    ENABLED: False
    NUM_ITER: 200
VERSION: 2
VIS_PERIOD: 0
```

Now we have to use `DefaultPredictor` with our configuration in parameter .

This class will initialize the model and download pretrained weights.

```
predictor = DefaultPredictor(cfg)
```

To predict an image :

```
# Make prediction
outputs = predictor(im)
```

```
outputs
```

```
{'instances': Instances(num_instances=17, image_height=480, image_width=640, fields=[pred_boxes: Box
es(tensor([[1.4324e+02, 2.4539e+02, 4.6495e+02, 4.7987e+02],
         [1.1441e+02, 2.6908e+02, 1.4962e+02, 3.9316e+02],
         [2.5247e+02, 1.6397e+02, 3.3820e+02, 4.1206e+02],
         [1.0697e-01, 2.7545e+02, 7.8442e+01, 4.7884e+02],
         [5.1929e+02, 2.8065e+02, 5.6126e+02, 3.4840e+02],
         [5.6006e+02, 2.7298e+02, 5.9637e+02, 3.7065e+02],
         [4.9831e+01, 2.7545e+02, 7.9595e+01, 3.4123e+02],
         [3.4256e+02, 2.5185e+02, 4.1623e+02, 2.7543e+02],
         [3.2863e+02, 2.3107e+02, 3.9233e+02, 2.5647e+02],
         [3.8529e+02, 2.7219e+02, 4.1352e+02, 3.0471e+02],
         [3.4631e+02, 2.6899e+02, 3.8501e+02, 2.9849e+02],
         [4.0797e+02, 2.7319e+02, 4.6071e+02, 3.6698e+02],
         [5.0778e+02, 2.6580e+02, 5.7188e+02, 2.9060e+02],
         [5.0673e+02, 2.8314e+02, 5.3047e+02, 3.4471e+02],
         [5.9504e+02, 2.8468e+02, 6.1118e+02, 3.1512e+02],
         [4.0487e+02, 2.7350e+02, 4.3312e+02, 3.2721e+02],
         [3.0615e+02, 1.4716e+02, 3.1520e+02, 1.6825e+02]], device='cuda:0')), scores: tensor([0.999
5, 0.9925, 0.9900, 0.9849, 0.9784, 0.9733, 0.9711, 0.9225, 0.9204,
         0.9150, 0.9014, 0.8770, 0.8623, 0.7257, 0.5874, 0.5068, 0.5018],
        device='cuda:0'), pred_classes: tensor([17,  0,  0,  0,  0,  0,  0, 25, 25,  0,  0,  0, 25,
0,  0,  0, 29],
        device='cuda:0')])}
```

```
outputs["instances"].to("cpu").pred_boxes
```

```
Boxes(tensor([[1.4324e+02, 2.4539e+02, 4.6495e+02, 4.7987e+02],
        [1.1441e+02, 2.6908e+02, 1.4962e+02, 3.9316e+02],
        [2.5247e+02, 1.6397e+02, 3.3820e+02, 4.1206e+02],
        [1.0697e-01, 2.7545e+02, 7.8442e+01, 4.7884e+02],
        [5.1929e+02, 2.8065e+02, 5.6126e+02, 3.4840e+02],
        [5.6006e+02, 2.7298e+02, 5.9637e+02, 3.7065e+02],
        [4.9831e+01, 2.7545e+02, 7.9595e+01, 3.4123e+02],
        [3.4256e+02, 2.5185e+02, 4.1623e+02, 2.7543e+02],
        [3.2863e+02, 2.3107e+02, 3.9233e+02, 2.5647e+02],
        [3.8529e+02, 2.7219e+02, 4.1352e+02, 3.0471e+02],
        [3.4631e+02, 2.6899e+02, 3.8501e+02, 2.9849e+02],
        [4.0797e+02, 2.7319e+02, 4.6071e+02, 3.6698e+02],
        [5.0778e+02, 2.6580e+02, 5.7188e+02, 2.9060e+02],
        [5.0673e+02, 2.8314e+02, 5.3047e+02, 3.4471e+02],
        [5.9504e+02, 2.8468e+02, 6.1118e+02, 3.1512e+02],
        [4.0487e+02, 2.7350e+02, 4.3312e+02, 3.2721e+02],
        [3.0615e+02, 1.4716e+02, 3.1520e+02, 1.6825e+02]]))
```

```
outputs["instances"].to("cpu").scores
```

```
tensor([0.9995, 0.9925, 0.9900, 0.9849, 0.9784, 0.9733, 0.9711, 0.9225, 0.9204,
        0.9150, 0.9014, 0.8770, 0.8623, 0.7257, 0.5874, 0.5068, 0.5018])
```

```
outputs["instances"].to("cpu").pred_classes
```

```
tensor([17,  0,  0,  0,  0,  0,  0, 25, 25,  0,  0,  0, 25,  0,  0,  0, 29])
```

`Visualizer` is a class that allow us to easily visualize the predictions and the results.

```
v = Visualizer(im[:, :, ::-1], MetadataCatalog.get(cfg.DATASETS.TRAIN[0]), scale=1.2)
v = v.draw_instance_predictions(outputs["instances"].to("cpu"))
```

```
plt.figure(figsize = (14, 10))
plt.imshow(cv2.cvtColor(v.get_image()[:, :, ::-1], cv2.COLOR_BGR2RGB))
```