NMS algorithm! (non maximum suppression)

Pre-processing step used in object detection to eliminate duplicate bounding boxes that predict the same object

- keep most confident Box.
- Remove all overlapping, lower-confidence ones.

How it works?

1. Sort all detected boxes by confidence score (↑ to ↓)
2. Select top box (highest confidence)
3. Remove all other boxes that overlap too much with it (using IoU - intersection over Union)
4. Repeat until no boxes remain!.

IoU threshold:

if 2 boxes have IoU > 0.5 → one with lower conf gets suppressed

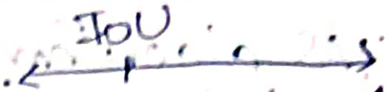
if IoU ≤ 0.5 → both boxes are kept.

Code:

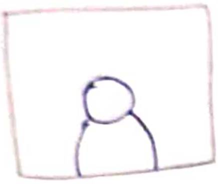
```
import cv2;
boxes = [[x, y, w, h], ...]
confidences = [0.9, 0.7, 0.6...]
indices = cv2.dnn.NMSBoxes(boxes, confidences,
                           score_threshold = 0.5,
                           nms_threshold = 0.4)

for i in indices:
    box = boxes[i]
```

IoU: Intersection Over Union

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

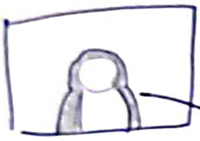


(dont overlap at all) 0 — 0.4 (threshold) — 1 (are the same)



OG img

→ IoU =  / $\boxed{8}$ (total).



pred

→ a bit more than OG image

i.e, intersection

Crack Detection 02/2 version. 2.0 [RoboFlow Instant 1 [EVAL]]

Roboflow 3.0 object detection.

dataset version

mAP@50: 64.2%

Precision: 64.8%

Recall: 59.1%

Source imgs:

IMGS: 1811	train: 1.5k
Classes: 1	test: 115
Unannotated: 882	valid: 197

Preprocessing:	Augmentations		Version size
auto-oriented	Flip	Blur	3210(2X)
grayscale	Crop		
Resize	Rotation		

```
import cv2
import numpy as np
import requests
<-- configuration -->
img_path = "..."
tile_size = 640
overlap = 0.2
api_key = "..."
model_id = "..."
confidence_threshold = 0.3
nms_iou_threshold = 0.4
<-- load image -->
image = cv2.imread(img_path)
height, width, _ = image.shape
```

← Tiling the image →

```
stride = int(tile_size * (1-overlap))
tiles = []

for y in range(0, height, stride):
    for x in range(0, width, stride):
        x_end = min(x + tile_size, width)
        y_end = min(y + tile_size, height)
        tile = image[y:y_end, x:x_end]
        tiles.append(((x,y), tile))
```

@opencv
img[h,w,ch]

← Inference Function →

```
def inference_tile(tile_img):
    _, img_encoded = cv2.imencode(".jpg", tile_img)

    try:
        response = requests.post(
            f"https://detect.roboflow.com/{model_id}?
            {api_key}&confidence={confidence_threshold}",
            files= {"file": img_encoded.tobytes()}
        )

        if response.status_code != 200:
            print("Error:" response.text)
            return []
        return response.json().get("predictions", [])

    except Exception as e:
        print("Request failed:", e)
        return []
```

Some Error Handling.

← Run Inference on all tiles →

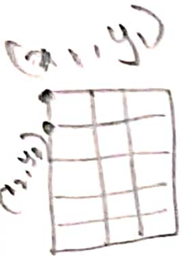
tiles = [ ] (ex)

tile((x, y), tile)

x-offset y-offset

((640,0), img)

array([[[0,0,0],
[0,0,0], ...
])




```
all_boxes = []
for (x-offset, y-offset), tile_img = tiles:
    preds = infer_tile(tile-img)
    for pred in preds:
        x1 = int(pred['x'] - pred['width']/2) +
                                  x-offset
        y1 = int (pred['y'] - pred['height'] /2) + y-offset
        x2 = int(pred ['x'] + pred ['width'] /2) + x-offset
        y2 = int(pred ['y'] + pred ['height']/2) + y-offset
        conf = pred ['confidence']
        all_boxes.append ([x1, y1, x2, y2, conf])
```

← Apply NMS →

```
if len (all-boxes) == 0:
    print("no detections found")
    exit()

boxes-np = np.array(all-boxes)
Boxes = boxes-np [:, :4].astype(int)
scores = boxes-np [:, :4].astype (float)
indices = cv2.dnn.NMSBoxes(  bboxes = boxes.tolist (),
                             scores = scores.tolist(),
                    score_threshold = confidence-threshold,
                    nms-threshold = nms-iou-threshold
          )
```

NEW FINDS !! (all actually !)

Final →

```
final_img = image.copy()
final_boxes = []
print("\n final detections (after NMS):")
for i in indices:
    i = i[0] if isinstance(i, (list, np.ndarray)) else i
    x1, y1, x2, y2 = boxes[i]
    final_boxes.append((x1, y1, x2, y2))
    print(f"box: ({x1}, {y1}, {x2}, {y2})")
    cv2.rectangle(final_image, (x1, y1), (x2, y2),
                                (0,0,0), thickness=10)

op_path = '.....'
cv2.imwrite(op_path, final_img)
print(f"Saved Result to:" op_path)
```

NEW FINDS ! (all actually !)

cv2.imread(img-path)

h, w, ch = img.shape

success, encoded_image = cv2.imencode(ext, image) → ".png" / ".jpg" ..etc

response = requests.post(f'url', data/json/files = ' ') → dict/file

(requests.Response)
response object

→ img_encoded.tobytes() // API's take byte data not numpy array/smtg !

error handling: if (response.status_code != 200) → safe/not

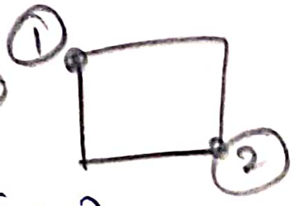
response.json().get("predictions", [ ])

indices = cv2.dnn.NMSBoxes(boxes,
confidences,
score_threshold = 0.5,
nms_threshold = 0.4)

(below this, remove them
(IoU overlap > 0.4 !

isinstance (?, (list, np.ndarray, int, ..)) → any you like/want

cv2.rectangle (final_img,
(x1, y1), 1
(x2, y2), 2
(0, 0, 0), → black (BGR)
thickness = 10) → (-1 = filled rectangle).



TILING IMAGES:-

Paths: INPUT_DIR! DATASET
LABEL_DIR! os.path.join(INPUT_DIR, "labels")
OUTPUT_IMG.DIR!. / / tiled ~~da~~dataset / img
OUTPUT_LABEL_DIR! . / . / tiled-dataset / labels

Tile_size = 640
Overlap = 0
STEP = TILE_STEP - OVERLAP **#stride**

O/P directories EXISTANCE CHECK! (// make)

```
os.makedir(OUTPUT_IMG_DIR, exist_ok=True)
os.makedir(OUTPUT_LABEL_DIR, exist_ok=True)
```

READ YOLO ANNOTATIONS → WRITE YOLO ANNOTATIONS

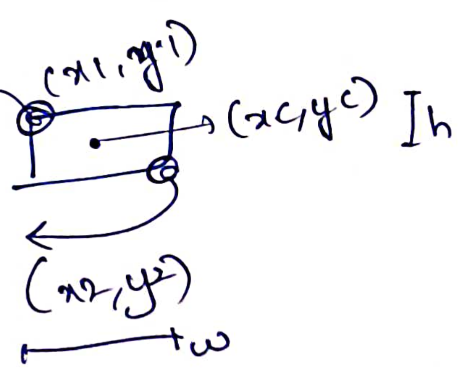
(reads .txt & returns boxes in pixel coordinates) | (take pixel coords & writes them back into YOLO format for each tile)

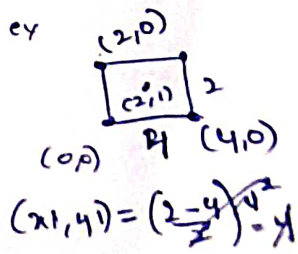
| G L O B A L | → | L O C A L |

```
def read_yolo_annotations(file_path, img_w, img_h):
    box = []
    if not os.path.exist(file_path):
        return boxes
    with open(file_path, 'r') as f:
        for line in f:
            parts = line.strip().split()
            if (len(parts)) = 5:
                cls, xc, yc, w, h = map(float, parts)
                x1 = (xc - w/2) * img_w
                y1 = (yc - h/2) * img_h
                x2 = (xc + w/2) * img_w
                y2 = (yc + h/2) * img_h
                boxes.append((int(cls), x1, y1, x2, y2))
    return boxes.
```

converts ~~absolute~~ normalized centers to absolute corner coordinates

(x1, y1) top left
(x2, y2) bottom right





```
def write_yolo_annotations(file_path, boxes, tile_w, tile_h):
    with open(file_path, 'w') as f:
        for cls, x1, y1, x2, y2 in boxes:
            xc = ((x1+x2)/2)/tile_w
            yc = ((y1+y2)/2)/tile_h
            w = (x2-x1)/tile_w
            h = (y2-y1)/tile_h
            if 0<=xc<=1 and 0<=yc<=1:
                f.write(f"{cls} {xc:.6f} {yc:.6f} {w:.6f}
                        {h:.6f}\n")
```

Converts absolute pixel coords back to normalized YOLO format

→ makes sure box is fully / mostly with in the tile

TILING CODE:-

TILE-SIZE = 640
(640 × 640)

```
def tile-image-and-labels (image-path):
    img = cv2.imread (image-path)   (read image)
    if img is None:
        print(f"Couldn't read img: {image-path}")   [if none return]
        return
[get just names]
    h, w = img.shape[:2]   [get h,w dimentions)
    fname = os.path.splitext (os.path.basename (image-path))[0]
    label-path = os.path.join (LABEL-DIR, fname + ".txt")
(get anno's)
    all_boxes = read-yolo-annotations (label-path, w, h)
    tile-count = 0   (initialize count).
[top R to bottom L]
    for y in range (0, h-TILE-SIZE+1, STEP):
        for x in range (0, w-TILE-SIZE + 1, STEP):
[CROP]
            tile = img [y : y+TILE_SIZE, x: x+TILE_SIZE]
[Put name save img patch]
            tile_filename = f"{name}_tile_{tile-count}.jpg"
            tile_path = os.path.join (OUTPUT-IMG-DIR, tile_filename)
            cv2.imwrite(tile_path, tile)

            tile_boxes = []   {Annots of patch [local annots]}
[How much of the obj lies in this tile ?]
            for cls, x1, y1, x2, y2 in all_boxes:   (x1,y1) -> top left   (_, _) -> bottom Right
                inter_x1 = max(x, x1)
                inter_y1 = max (y, y1)
                inter_x2 = min (x+TILE_SIZE, x2)
                inter_y2 = min (y+TILE_SIZE, y2)
(intersection box)
[if intersection there] & (they overlap)
                if inter_x1 < inter_x2 and inter_y1 < inter_y2:
• obj to tile local coords
                    adj_x1 = inter_x1 - x
                    adj_y1 = inter_y1 - y
                    adj_x2 = inter_x2 - x
                    adj_y2 = inter_y2 - y
• annot recentered inside tile
• append in tile_boxes list
                    tile_boxes.append((cls, adj_x1, adj_y1, adj_x2,
                                       adj_y2))
```

```
tile_label-path = os.path.join(OUTPUT_LABEL_DIR,
                    tile-filename.replace(".jpg", ".txt
```
(replace the annotations)

```
write-yolo_annotations(tile_label_path, tile-boxes, TILE_SIZE,
                        TILE_SIZE)
```
(rewrite the annotations)

```
tile_count += 1
```
(update count)

```
print(f" tile {fname} into {tile_count} patches.")
```

```
# for fname in os.listdir(INPUT_DIR):
      if fname.lower().endswith((".jpg", ".jpeg", ".png")):
          tile_image_and_labels(os.path.join(INPUT_DIR, fname))
```

(loop through all the images)

If not work else needs improvments

- Padding (instead of skipping small edges)
- Add overlap (strides) (25%) ...

NMS (non-maximum suppression) → pre-processing step used in object detection to eliminate duplicate bounding boxes that predict the same object.

- keep the most confident box
- remove all overlapping, lower-confidence ones.