Task: Custom Object Detection and Novel Bounding Box Metric with YOLO

By P.K. Udith Ishanka Sandaruwan

1. Overview

- Dataset used - Oxford-IIIT Pet Dataset
- Used the yolov5 official repo as the code base.
- My work is committed in this repo - udithishanka/custom_bounding_box

2. Setup YOLO

I used the YOLOv5 official repository code with the Oxford-IIIT Pet Dataset. First, I downloaded the dataset, which includes images and Pascal VOC-style XML annotations. Then, I wrote a custom script that converts the annotations from Pascal VOC XML format to YOLO-compatible format.

The script parses each XML file, extracts the object bounding box coordinates, normalizes them according to the image size, and saves them as .txt files in the YOLO format (class_id, x_center, y_center, width, height). The images and corresponding annotations were then split into training and validation sets, with 80% allocated for training and 20% for validation. The preprocess.py script handles the conversion of XML files to the YOLO format and organizes the dataset into respective train and validation folders for further training with YOLOv5.

3. Custom Bounding Box Similarity Metric

- **IoU (Intersection over Union)** - used to measure the overlap between two bounding boxes. A high IoU means the two boxes are highly similar, and a low IoU means they are dissimilar or do not overlap.

$$IoU = \frac{Area\ of\ Overlap}{Area\ of\ Union}$$

  Where:
  - Area of Overlap is the area of intersection between the two boxes.
  - Area of Union is the total area covered by both boxes, which is the sum of the individual areas minus the overlap.

- **Aspect Ratio Similarity (ARS) -** Aspect ratio similarity measures how similar the aspect ratios (width/height) of two bounding boxes are. A mismatch in aspect ratios suggests that one box is more compact compared to the other, that may indicate a poor fit for the object.

$$ARS = 1 - \frac{\left|\frac{w1}{h1} - \frac{w2}{h2}\right|}{\frac{w2}{h2} + \epsilon}$$

Where:

- w1 and h1 are the width and height of box1, respectively.
- w2 and h2 are the width and height of box2, respectively.
- ϵ\epsilonϵ is a small constant to avoid division by zero (to prevent instability when w2 or h2 is too small).

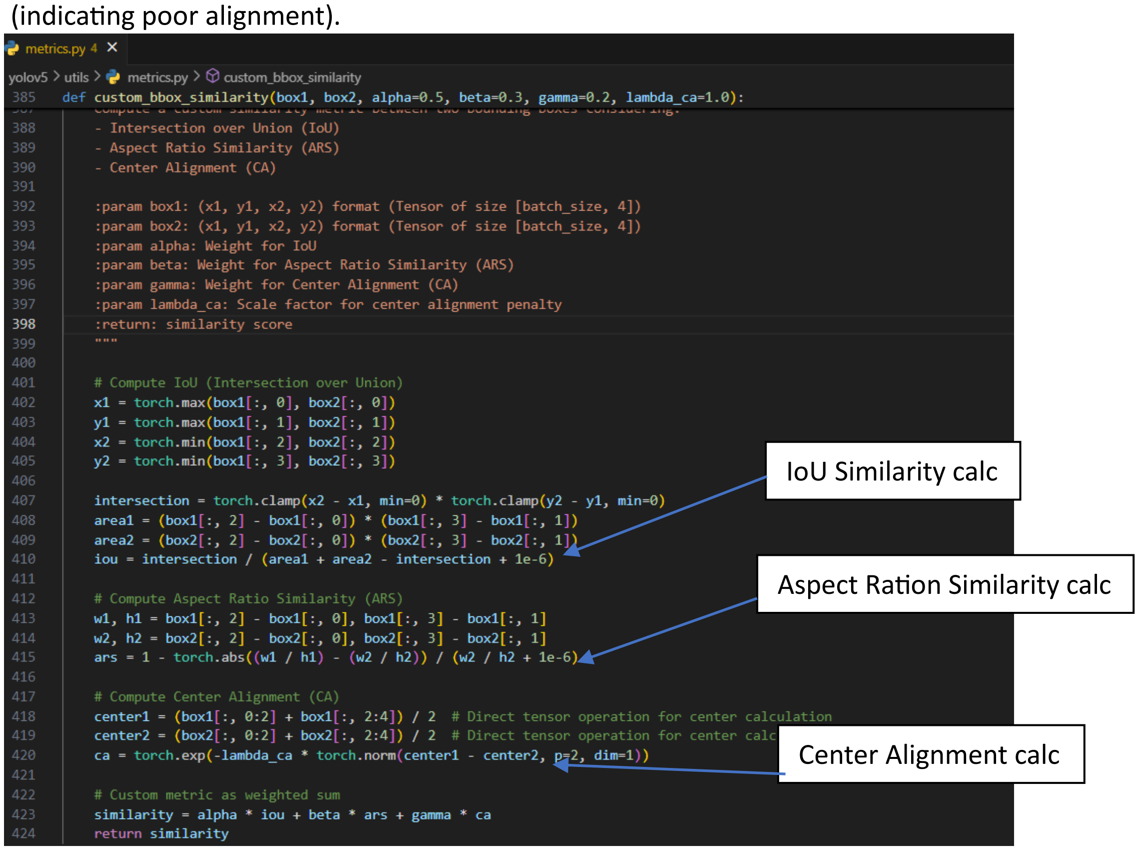
Purpose - **Penalizes mismatches in box proportions**: The aspect ratio similarity penalizes bounding boxes where the width-to-height ratio is significantly different, indicating that the boxes are of different shapes.

- **Center Alignment (CA) -** Center alignment measures how well the centers of two bounding boxes are aligned with each other. It calculates the distance between the centers and penalizes large shifts in position.

$$CA = e^{-\lambda||(x1,y1)-(x2-y2)||2}$$

Purpose - Penalizes shifts in the predicted box: Center alignment focuses on how well the predicted bounding box is positioned relative to the ground truth box. If the predicted box is far from the true box (in terms of its center), the penalty increases.

If the predicted box's center is close to the ground truth's center, the CA score will be high (indicating good alignment).
If the predicted box is far from the ground truth's center, the CA score will be low (indicating poor alignment).



```python
def custom_bbox_similarity(box1, box2, alpha=0.5, beta=0.3, gamma=0.2, lambda_ca=1.0):
    - Intersection over Union (IoU)
    - Aspect Ratio Similarity (ARS)
    - Center Alignment (CA)

    :param box1: (x1, y1, x2, y2) format (Tensor of size [batch_size, 4])
    :param box2: (x1, y1, x2, y2) format (Tensor of size [batch_size, 4])
    :param alpha: Weight for IoU
    :param beta: Weight for Aspect Ratio Similarity (ARS)
    :param gamma: Weight for Center Alignment (CA)
    :param lambda_ca: Scale factor for center alignment penalty
    :return: similarity score
    """

    # Compute IoU (Intersection over Union)
    x1 = torch.max(box1[:, 0], box2[:, 0])
    y1 = torch.max(box1[:, 1], box2[:, 1])
    x2 = torch.min(box1[:, 2], box2[:, 2])
    y2 = torch.min(box1[:, 3], box2[:, 3])

    intersection = torch.clamp(x2 - x1, min=0) * torch.clamp(y2 - y1, min=0)
    area1 = (box1[:, 2] - box1[:, 0]) * (box1[:, 3] - box1[:, 1])
    area2 = (box2[:, 2] - box2[:, 0]) * (box2[:, 3] - box2[:, 1])
    iou = intersection / (area1 + area2 - intersection + 1e-6)

    # Compute Aspect Ratio Similarity (ARS)
    w1, h1 = box1[:, 2] - box1[:, 0], box1[:, 3] - box1[:, 1]
    w2, h2 = box2[:, 2] - box2[:, 0], box2[:, 3] - box2[:, 1]
    ars = 1 - torch.abs((w1 / h1) - (w2 / h2)) / (w2 / h2 + 1e-6)

    # Compute Center Alignment (CA)
    center1 = (box1[:, 0:2] + box1[:, 2:4]) / 2  # Direct tensor operation for center calculation
    center2 = (box2[:, 0:2] + box2[:, 2:4]) / 2  # Direct tensor operation for center calc
    ca = torch.exp(-lambda_ca * torch.norm(center1 - center2, p=2, dim=1))

    # Custom metric as weighted sum
    similarity = alpha * iou + beta * ars + gamma * ca
    return similarity
```

Above code can be found in yolov5/utils/metrics.py.

I used a weighted sum to combine these factors, allowing the flexibility to adjust their relative importance through parameters such as alpha, beta, gamma, and lambda_ca. This custom similarity metric is useful for fine-tuning the detection of objects with similar shapes and positions. I have set alpha, beta, gamma as fixed values, we can do a hyper parameter tuning to find out what are the best values for them.

4. **Incorporate Your Metric into the Training or Evaluation Loop**

```
loss.py 2 ×
yolov5 > utils > loss.py > ComputeLoss > _call_
class ComputeLoss:
    def __init__(self, model, autobalance=False, lambda_factor=0.0):
        self.nl = m.nl  # number of layers
        self.anchors = m.anchors
        self.device = device
        self.lambda_factor = lambda_factor

    def __call__(self, p, targets):  # predictions, targets
        """Performs forward pass, calculating class, box, and object loss for given predictions and targets."""
        lcls = torch.zeros(1, device=self.device)  # class loss
        lbox = torch.zeros(1, device=self.device)  # box loss
        lobj = torch.zeros(1, device=self.device)  # object loss
        tcls, tbox, indices, anchors = self.build_targets(p, targets)  # targets

        # Losses
        for i, pi in enumerate(p):  # layer index, layer predictions
            b, a, gj, gi = indices[i]  # image, anchor, gridy, gridx
            tobj = torch.zeros(pi.shape[:4], dtype=pi.dtype, device=self.device)  # target obj

            if n := b.shape[0]:
                # pxy, pwh, _, pcls = pi[b, a, gj, gi].tensor_split((2, 4, 5), dim=1)  # faster, requires torch 1.8.0
                pxy, pwh, _, pcls = pi[b, a, gj, gi].split((2, 2, 1, self.nc), 1)  # target-subset of predictions

                # Regression
                pxy = pxy.sigmoid() * 2 - 0.5
                pwh = (pwh.sigmoid() * 2) ** 2 * anchors[i]
                pbox = torch.cat((pxy, pwh), 1)  # predicted box
                iou = bbox_iou(pbox, tbox[i], CIoU=True).squeeze()  # iou(prediction, target)

                custom_similarity = custom_bbox_similarity(pbox, tbox[i])

                lbox += (1.0 - iou).mean()  + self.lambda_factor*(1-custom_similarity).mean()

                # Objectness
                iou = iou.detach().clamp(0).type(tobj.dtype)
                if self.sort_obj_iou:
                    j = iou.argsort()
                    b, a, gj, gi, iou = b[j], a[j], gj[j], gi[j], iou[j]
                if self.gr < 1:
```

I added the new similarity metric **as an additional term.** I introduced a **lambda_factor as an argument which can be parsed when running the train.py script. Default value of lambda_factor is Zero. For each lambda values, we can see how the evaluation results change.** I added it as (1-custom_similarity) because, by subtracting it from 1, we're effectively treating higher similarity as a lower loss (i.e., more similar bounding boxes should reduce the loss).

```
py 4    train.py 7 ×
train.py > ...
ef parse_opt(known=False):
    parser.add_argument("--local_rank", type=int, default=-1, help="Automatic DDP Multi-GPU argument, do not modify")

    # Logger arguments
    parser.add_argument("--entity", default=None, help="Entity")
    parser.add_argument("--upload_dataset", nargs="?", const=True, default=False, help='Upload data, "val" option')
    parser.add_argument("--bbox_interval", type=int, default=-1, help="Set bounding-box image logging interval")
    parser.add_argument("--artifact_alias", type=str, default="latest", help="Version of dataset artifact to use")

    # NDJSON logging
    parser.add_argument("--ndjson-console", action="store_true", help="Log ndjson to console")
    parser.add_argument("--ndjson-file", action="store_true", help="Log ndjson to file")
    parser.add_argument("--lambda_factor", type=float, default=0.0, help="Lambda parameter for custom bounding box loss")

    return parser.parse_known_args()[0] if known else parser.parse_args()
```

```
metrics.py 4    train.py 7    loss.py 2 ×
ov5 > utils > loss.py > ComputeLoss
class QFocalLoss(nn.Module):
    def forward(self, pred, true):
            return loss


class ComputeLoss:
    """Computes the total loss for YOLOv5 model predictions, including classification, box, and objectness losses."""

    sort_obj_iou = False

    # Compute losses
    def __init__(self, model, autobalance=False, lambda_factor=0.0):
        """Initializes ComputeLoss with model and autobalance option, autobalances losses if True."""
        device = next(model.parameters()).device  # get model device
        h = model.hyp  # hyperparameters

        # Define criteria
        BCEcls = nn.BCEWithLogitsLoss(pos_weight=torch.tensor([h["cls_pw"]], device=device))
        BCEobj = nn.BCEWithLogitsLoss(pos_weight=torch.tensor([h["obj_pw"]], device=device))

        # Class label smoothing https://arxiv.org/pdf/1902.04103.pdf eqn 3
        self.cp, self.cn = smooth_BCE(eps=h.get("label_smoothing", 0.0))  # positive, negative BCE targets

        # Focal loss
        g = h["fl_gamma"]  # focal loss gamma
        if g > 0:
            BCEcls, BCEobj = FocalLoss(BCEcls, g), FocalLoss(BCEobj, g)

        m = de_parallel(model).model[-1]  # Detect() module
        self.balance = {3: [4.0, 1.0, 0.4]}.get(m.nl, [4.0, 1.0, 0.25, 0.06, 0.02])  # P3-P7
        self.ssi = list(m.stride).index(16) if autobalance else 0  # stride 16 index
        self.BCEcls, self.BCEobj, self.gr, self.hyp, self.autobalance = BCEcls, BCEobj, 1.0, h, autobalance
        self.na = m.na  # number of anchors
        self.nc = m.nc  # number of classes
        self.nl = m.nl  # number of layers
        self.anchors = m.anchors
        self.device = device
        self.lambda_factor = lambda_factor
```

lambda_factor taken as an argument to the ComputeLoss class.

5. Experimental Results and Analysis

First, let's try to run a training without parsing a lambda_factor, which means by default lambda_factor is zero, and loss function will not have an additional term. I am using yolov5s pretrained weights, for every evaluation, and training for 10 epochs (Accuracy will be higher if we train for more epochs but my focus was to evaluate how the additional loss term will affect the accuracy, with same number of epochs, with different lambda factors).

- Lambda_factor = 0 (Training without custom bounding box similarity

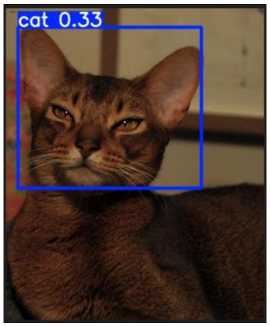
python train.py --img 640 --batch 16 --epochs 10 --data data/cats_dogs.yaml --weights yolov5s.pt

```
Eval results - without the custom bounding box

                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100%|██████████| 93/93 [00:08<00:00, 10.62it/s]
                   all       1478        749      0.477      0.932      0.522      0.409
                   cat       1478        238       0.48      0.945      0.531      0.441
                   dog       1478        511      0.474      0.918      0.513      0.377
Speed: 0.1ms pre-process, 2.4ms inference, 1.0ms NMS per image at shape (16, 3, 640, 640)
Results saved to runs/val/exp2
```

Use this command to test some sample images in the validation dir. **New trained weights path** - runs/train/exp4/weights/best.pt, **test image_path** - ../datasets/cats_dogs/images/val/Abyssinian_6.jpg

python detect.py --weights runs/train/exp4/weights/best.pt --source ../datasets/cats_dogs/images/val/Abyssinian_6.jpg --img 640 --conf 0.25

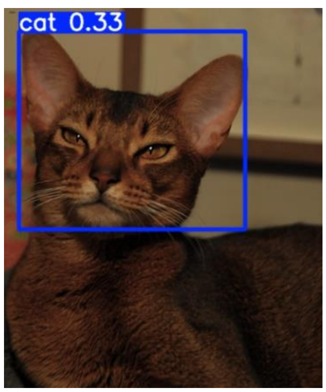


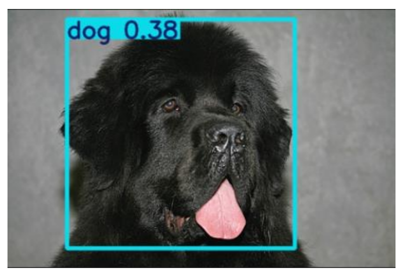
- lambda_factor = 0.01

```
Eval results - with the custom bounding box (lambda_factor = 0.01)
Model summary: 157 layers, 7015519 parameters, 0 gradients, 15.8 GFLOPs
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100%|██████████| 47/47 [00:06<00:00,  6.88it/s]
                   all       1478        749      0.476      0.934      0.512      0.401
                   cat       1478        238      0.476      0.946      0.529      0.441
                   dog       1478        511      0.475      0.922      0.496       0.36
Results saved to runs/train/exp13
```

python train.py --img 640 --batch 16 --epochs 10 --data data/cats_dogs.yaml --weights yolov5s.pt –lambda_factor 0.01

when we are using lambda_factor – 0.01, there seems to be an increase in Recall values, however Mean Average Precision (mAP) has degraded slightly. Precision values are around the same.



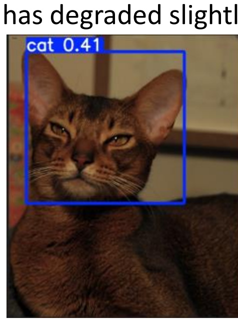


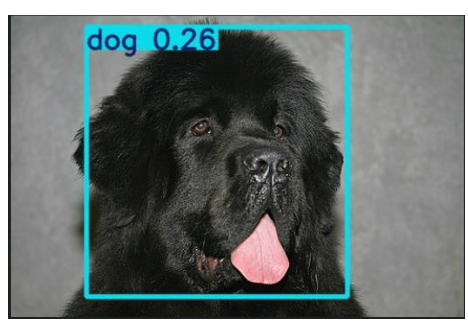
- lambda_factor = 0.05

```
Eval results - with the custom bounding box (lambda_factor = 0.05)
Model summary: 157 layers, 7015519 parameters, 0 gradients, 15.8 GFLOPs
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100%|██████████| 47/47 [00:06<00:00,  6.90it/s]
                   all       1478        749      0.477      0.944      0.499      0.381
                   cat       1478        238      0.468      0.966      0.494      0.408
                   dog       1478        511      0.487      0.922      0.503      0.355
Results saved to runs/train/exp14
```

python train.py --img 640 --batch 16 --epochs 10 --data data/cats_dogs.yaml --weights yolov5s.pt –lambda_factor 0.05

when using lambda_factor - 0.05, Recall accuracies have increased significantly, some of the Precision values has increased as well. mAP values has degraded slightly.



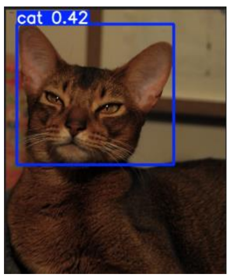


- lambda_factor = 0.1

```
Eval results - with the custom bounding box (lambda_factor = 0.1)
Model summary: 157 layers, 7015519 parameters, 0 gradients, 15.8 GFLOPs
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100%|██████████| 47/47 [00:06<00:00,  7.19it/s]
                   all       1478        749      0.414      0.714      0.451      0.285
                   cat       1478        238      0.365      0.878       0.48      0.329
                   dog       1478        511      0.464       0.55      0.422      0.241
Results saved to runs/train/exp15
```

python train.py --img 640 --batch 16 --epochs 10 --data data/cats_dogs.yaml --weights yolov5s.pt –lambda_factor 0.1

when using lambda_factor as 0.1 , performance has degraded in almost all of the metrics, compared to without using any custom bounding box similarity.

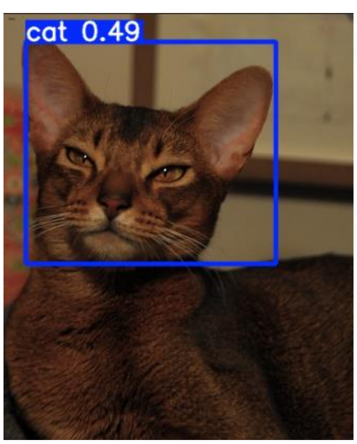


- lambda_factor = 0.5

```
Eval results - with the custom bounding box (lambda_factor = 0.5)
Model summary: 157 layers, 7015519 parameters, 0 gradients, 15.8 GFLOPs
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100%|██████████| 47/47 [00:06<00:00,  6.89it/s]
                   all       1478        749      0.483      0.916      0.495      0.381
                   cat       1478        238      0.485      0.927        0.5      0.407
                   dog       1478        511       0.48      0.904      0.491      0.354
Results saved to runs/train/exp16
```

python train.py --img 640 --batch 16 --epochs 10 --data data/cats_dogs.yaml --weights yolov5s.pt –lambda_factor 0.5

when using lambda_factor as 0.5, results were comparably better than using lambda_factor as 0.1. so that concludes, performance doesn’t change linearly with the lambda_factor. There’s definitely a sweet spot, and we have to find it with a hyper parameter tuning.



No detections

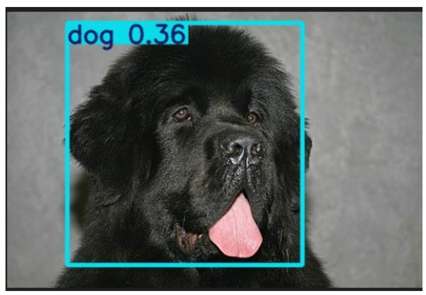
- lambda_factor = 1.0

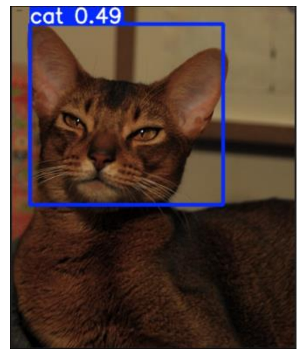
```
Eval results - with the custom bounding box (lambda_factor = 1.0)
Model summary: 157 layers, 7015519 parameters, 0 gradients, 15.8 GFLOPs
                 Class     Images  Instances          P          R      mAP50   mAP50-95: 100%|██████████| 47/47 [00:06<00:00,  7.12it/s]
                   all       1478        749      0.416      0.656      0.409      0.263
                   cat       1478        238      0.384      0.802      0.436      0.309
                   dog       1478        511      0.448      0.511      0.382      0.217
Results saved to runs/train/exp17
```

python train.py --img 640 --batch 16 --epochs 10 --data data/cats_dogs.yaml --weights yolov5s.pt –lambda_factor 1.0

using lambda_factor = 1.0, performance has degraded compared to every other secanario.

Out of these, iterations, lambda_factor = 0.05 performed the best.





6. Reflective Questions

Performance: Did your custom similarity metric improve or degrade performance (either qualitatively or quantitatively)?

As you can see in Experimental Results section, for some lambda_factors it degraded the performance, but lambda_factor 0.01, 0.05 performed better than using only IoU similarity.

Although when lambda_factor 0.01, 0.05 , there was a significant increase in Recall, and Precision values, performance slightly degraded according to mAP50 and mAP50-95.

Trade-offs: Discuss any computational or conceptual trade-offs of your metric vs. standard IoU-based metrics.

- **Computational**

The custom similarity metric that I introduced, didn't change much in the required computational power.

To finetune a YOLOv5, without a custom bounding box required around 5.8GB of memory, and it was around the same, when I introduced my own custom bounding box.

My metric is still O(1) per bounding box pair, making it computationally feasible for large-scale object detection. However, the additional computations (aspect ratio difference and center distance penalty) slightly require more computation than IoU alone.

- **Conceptual**

  Advantages

  Standard IoU only considers area overlap. ARS ensures that bounding boxes with very different proportions receive a lower similarity score. This might be important for detecting objects with distinct aspect ratios (e.g., a tall bottle vs. a wide pan).

  IoU can be misleading when two boxes are close but do not overlap significantly (e.g., an off-center prediction). CA explicitly penalizes predictions that are shifted away from the target, even if they have similar size and shape.

  Limitations

  - More Parameters to Tune.
  - My metric calculates the similarity with three factors combined, we need to know much exactly each factor contributes to the similarity. Too many hyperparameters to tune.

Future Ideas:

- We can do adaptive scaling for center alignment, instead of a fixed λ, dynamically adjust it based on object size.

  $$CA = e^{-\lambda||(x1,y1)-(x2-y2)||2}$$

  smaller object should have a higher CA penalty, larger ones should have a smaller penalty.
- Distance-Based Penalty for Non-Overlapping Boxes - Introduce a **soft penalty** for near-misses, Instead of **IoU = 0** when there's no overlap, introduce a **Gaussian falloff function** based on the **Euclidean distance** between the boxes.