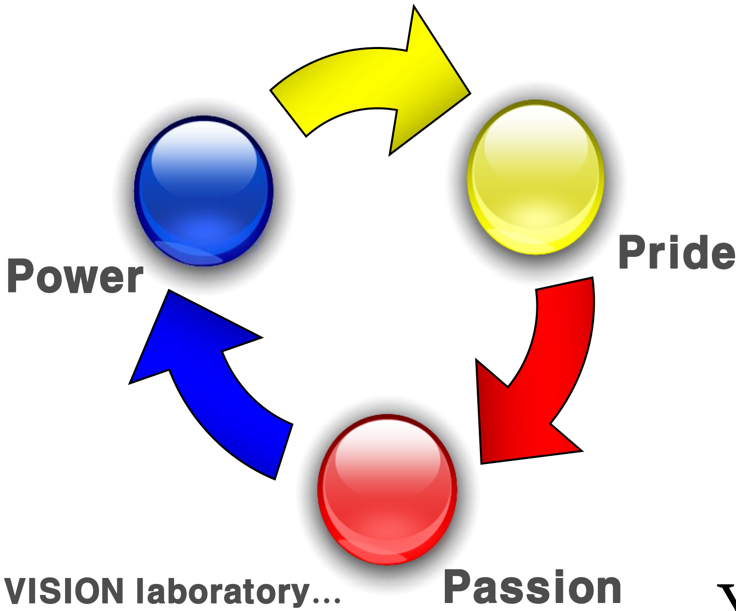


P³ in VISION laboratory…

You Only Look Once:
Unified, Real-Time Object Detection
+ code

Vision System Lab, Gyumin Park
yywnnaa@gmail.com
Feb 02, 2024

Soongsil University
VISION SYSTEM LABOARTORY

Abstract

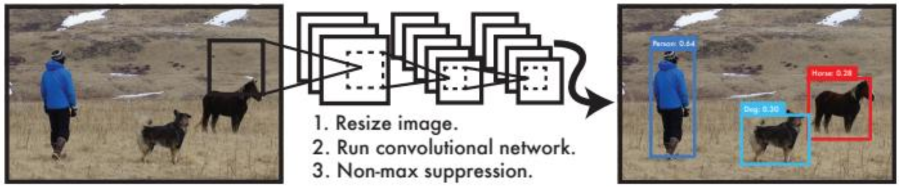


Figure 1: The YOLO Detection System. Processing images with YOLO is simple and straightforward. Our system (1) resizes the input image to 448×448, (2) runs a single convolutional network on the image, and (3) thresholds the resulting detections by the model's confidence.

- you only look once (YOLO) at an image to predict what objects are present and where they are
- refreshingly simple
- single neural network
- extremely fast
- more localization errors
- less likely to predict false positives on background

1. Introduction

several benefits over traditional methods

1. extremely fast

achieves high mAP

2. reasons globally about the image when making predictions (next slide)

sees the entire image during training and test time

makes less than half the number of background errors compared to Fast R-CNN

3. learns generalizable representations of objects

highly generalizable

less likely to break down when applied to new domains or unexpected inputs

less accuracy than sota

quickly identify objects

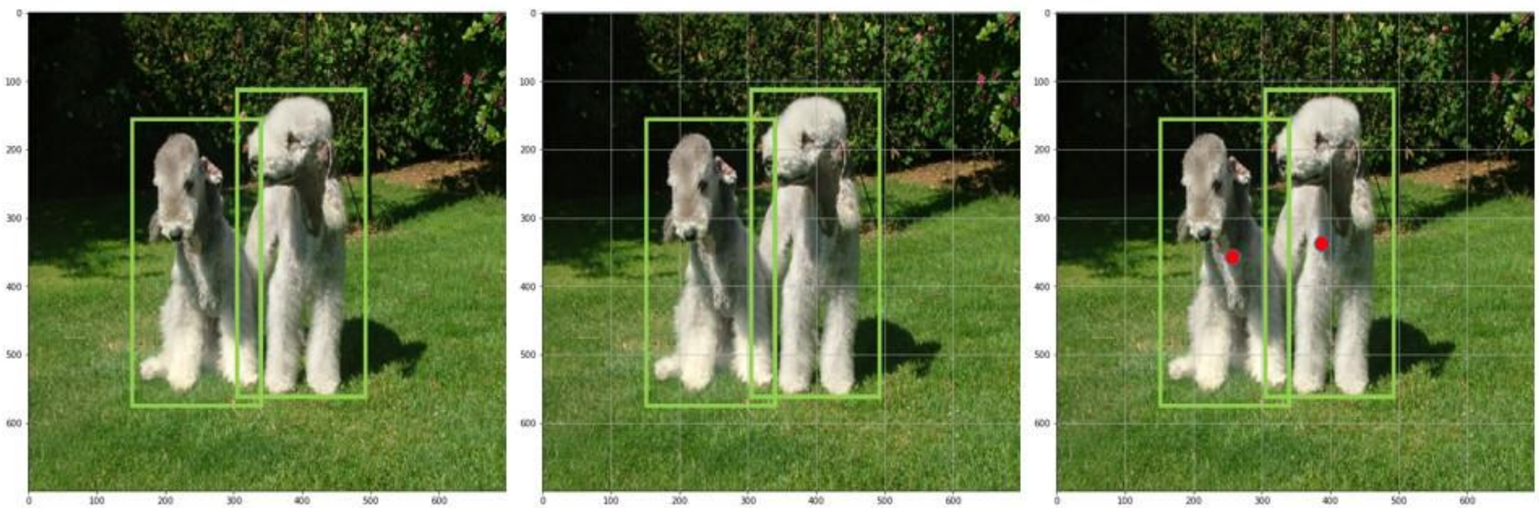
struggles to precisely localize some objects, especially small ones

2. Unified Detection

- single neural network object detection
- uses features from the entire image to predict each bounding box
- predicts all bounding boxes across all classes for an image simultaneously
- reasons globally about the full image and all the objects in the image
- real time speeds
- high average precision

VISION
SYSTEM
LABORATORY

2. Unified Detection



- **divides the input image into an S × S grid**
- If the center of an object falls into a grid cell, that grid cell is **responsible** for detecting that object

```
grid_size_x = data.size(dim=2) / config.S
grid_size_y = data.size(dim=1) / config.S
```

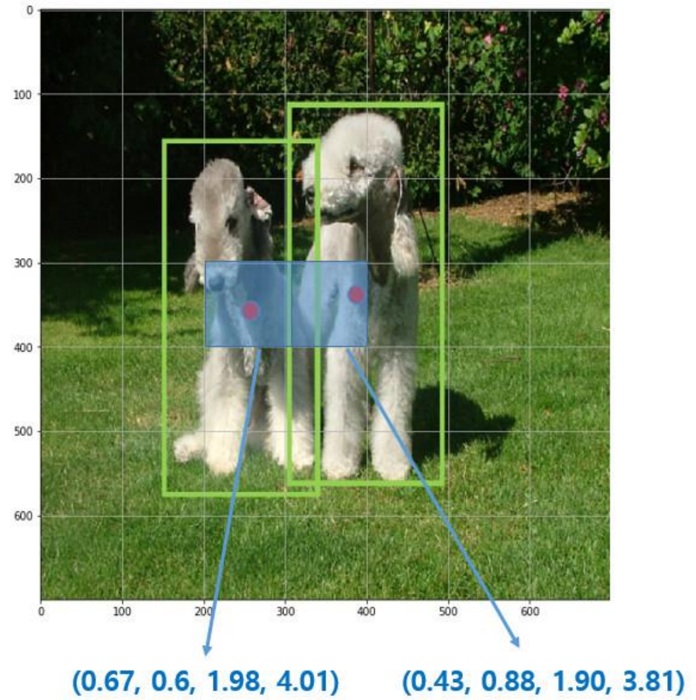
data.size(dim=2) 이미지의 가로 크기

data.size(dim=1) 이미지의 세로 크기

이를 통해 bounding box의 중심 좌표를 grid cell로 변환, ground truth tensor를 구성

VISION SYSTEM LABORATORY

2. Unified Detection



- Each bounding box consists of 5 predictions:
- x, y, w, h, and confidence

- (x, y) : represent the center of the box relative to the bounds of the grid cell
- width, height : predicted relative to the whole image

- confidence prediction : represents the IOU between the predicted box and any ground truth box

- Each grid cell predicts C conditional class probabilities (conditioned on the grid cell containing an object)
- only predict one set of class probabilities per grid cell, regardless of the number of boxes B

2. Unified Detection

```
# Calculate the position of center of bounding box
mid_x = (x_max + x_min) / 2
mid_y = (y_max + y_min) / 2
col = int(mid_x // grid_size_x)
row = int(mid_y // grid_size_y)
```

- mid_x, mid_y : bounding box의 중심 좌표
- 이 좌표를 grid_size_x와 grid_size_y로 나누면 bounding box가 속한 grid cell의 위치 col, row

2. Unified Detection

ground truth tensor

```
if 0 <= col < config.S and 0 <= row < config.S: # 그리드의 범위 안에 해당하는 경우
    cell = (row, col)
    if cell not in class_names or name == class_names[cell]:
        # 이미 할당된 클래스가 없거나 현재 클래스와 일치하는 경우
        # 해당 클래스에 대한 one-hot encoding을 ground truth tensor의 적절한 위치에 삽입
        # Insert class one-hot encoding into ground truth
        one_hot = torch.zeros(config.C)
        one_hot[class_index] = 1.0
        ground_truth[row, col, :config.C] = one_hot
        class_names[cell] = name

        # Insert bounding box into ground truth tensor
        bbox_index = boxes.get(cell, 0)
        if bbox_index < config.B: #  현재 셀에 할당된 bounding box의 수가 config.B 미만인 경우
            bbox_truth = ( #  bounding box의 정보를 one-hot 인코딩된 형태로 ground truth tensor에 추가
                (mid_x - col * grid_size_x) / config.IMAGE_SIZE[0], # X coord relative to grid square
                # bounding box 중심의 x 좌표를 그리드 셀 내에서 상대적인 위치로 변환
                (mid_y - row * grid_size_y) / config.IMAGE_SIZE[1], # Y coord relative to grid square

                (x_max - x_min) / config.IMAGE_SIZE[0],             # Width
                # bounding box의 가로 크기를 이미지 가로 크기로 정규화
                (y_max - y_min) / config.IMAGE_SIZE[1],             # Height
                # bounding box의 세로 크기를 이미지 세로 크기로 정규화
                1.0   # Confidence # bounding box의 존재 여부
            )

            # Fill all bbox slots with current bbox (starting from current bbox slot, avoid overriding prev)
            # This prevents having "dead" boxes (zeros) at the end, which messes up IOU loss calculations
            bbox_start = 5 * bbox_index + config.C # 현재 객체의 bounding box가 채워질 시작 인덱스 계산
            ground_truth[row, col, bbox_start:] = torch.tensor(bbox_truth).repeat(config.B - bbox_index)
            # 현재 bounding box의 정보 bbox_truth를 config.B-bbox_index번 복제하여 뒷 부분의 빈 bounding box 슬롯을 채움
            # 복제 안 하고 해당 슬롯만 채워지면 -> 나머지는 0으로 남음, 이후의 IOU 손실(IOU loss) 계산 시에 문제
            boxes[cell] = bbox_index + 1
```

2. Unified Detection

```python
def bbox_to_coords(t):
    """Changes format of bounding boxes from
    [x, y, width, height] to ([x1, y1], [x2, y2])."""

    # bounding box의 너비와 중심점의 x 좌표를 추출
    width = bbox_attr(t, 2)
    x = bbox_attr(t, 0)
    x1 = x - width / 2.0 # 중심점 x를 기준으로 왼쪽으로 얼마만큼 떨어져 있는지
    x2 = x + width / 2.0 # 중심점 x를 기준으로 오른쪽으로 얼마만큼 떨어져 있는지

    height = bbox_attr(t, 3)
    y = bbox_attr(t, 1)
    y1 = y - height / 2.0
    y2 = y + height / 2.0

    # x 좌표와 y 좌표를 나타내는 두 개의 텐서를 합치는 연산
    # torch.stack((x1, y1), dim=4)는 bounding box의 왼쪽 상단 모서리 좌표를 합쳐서 새로운 차원을 생성
    # torch.stack((x2, y2), dim=4)는 bounding box의 오른쪽 하단 모서리 좌표를 합쳐서 새로운 차원을 생성
    #  최종적으로 (batch, S, S, B, 2) 형태의 텐서가 생성
    return torch.stack((x1, y1), dim=4), torch.stack((x2, y2), dim=4)
```

- bounding box의 정보를 [x1, y1]과 [x2, y2] 형태로 변환

VISION SYSTEM LABORATORY

2. Unified Detection

iou

```python
def get_iou(p, a): #
    p_tl, p_br = bbox_to_coords(p)          # (batch, S, S, B, 2)
    # p의 top-left왼쪽 상단과 bottom-right오른쪽 하단 모서리 좌표
    a_tl, a_br = bbox_to_coords(a)
    # a의 왼쪽 상단과 오른쪽 하단 모서리 좌표

    # Largest top-left corner and smallest bottom-right corner give the intersection
    coords_join_size = (-1, -1, -1, config.B, config.B, 2)
    # torch.max 및 torch.min 연산에서 사용되는 두 텐서를 결합하기 위한 크기
    # -1은 해당 차원의 크기를 입력 텐서의 크기로 맞추라는 의미
    # (batch, S, S, B, B, 2) 모양의 텐서를 만들기 위한 크기를 가지고 있음

    tl = torch.max( # 두 bounding box의 top-left 좌표 중에서 더 큰 값을 선택
        p_tl.unsqueeze(4).expand(coords_join_size),         # (batch, S, S, B, 1, 2) -> (batch, S, S, B, B, 2)
        a_tl.unsqueeze(3).expand(coords_join_size)          # (batch, S, S, 1, B, 2) -> (batch, S, S, B, B, 2)
    )
    br = torch.min( # 두 bounding box의 bottom-right 좌표 중에서 더 작은 값을 선택
        p_br.unsqueeze(4).expand(coords_join_size),
        a_br.unsqueeze(3).expand(coords_join_size)
    )
    # -> 두 bounding box를 감싸는 가장 작은 사각형의 좌표를 얻음

    intersection_sides = torch.clamp(br - tl, min=0.0) # torch.clamp : 텐서의 값을 주어진 최소 및 최대 값으로 제한
    # br - tl : 두 바운딩 박스의 각 변에 대해 겹치는 길이 - > 각 변의 겹치는 길이를 0 이상으로 만든다
    intersection = intersection_sides[..., 0] \
                   * intersection_sides[..., 1]       # (batch, S, S, B, B)
    # 마지막 두 차원에 대한 요소곱 (두 바운딩 박스의 겹치는 부분의 너비와 높이)

    p_area = bbox_attr(p, 2) * bbox_attr(p, 3)                  # (batch, S, S, B)
    # 바운딩 박스의 폭(width, index 2)과 높이(height, index 3)를 곱하여 각 바운딩 박스의 면적을 계산
    p_area = p_area.unsqueeze(4).expand_as(intersection)        # (batch, S, S, B, 1) -> (batch, S, S, B, B)
    # 텐서 모양 맞추기 위해 확장

    a_area = bbox_attr(a, 2) * bbox_attr(a, 3)                  # (batch, S, S, B)
    a_area = a_area.unsqueeze(3).expand_as(intersection)        # (batch, S, S, 1, B) -> (batch, S, S, B, B)

    union = p_area + a_area - intersection

    # Catch division-by-zero
    zero_unions = (union == 0.0)
    union[zero_unions] = config.EPSILON
    intersection[zero_unions] = 0.0

    return intersection / union
```

2. Unified Detection

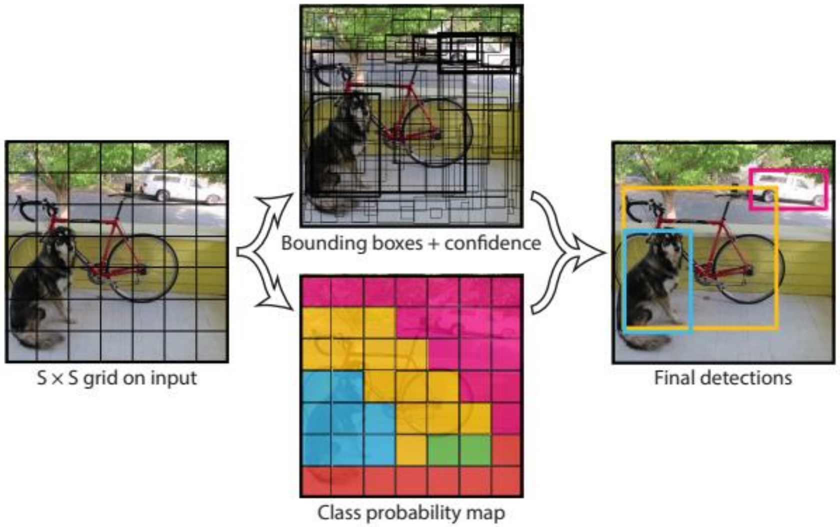


Figure 2: The Model. Our system models detection as a regression problem. It divides the image into an $S \times S$ grid and for each grid cell predicts B bounding boxes, confidence for those boxes, and C class probabilities. These predictions are encoded as an $S \times S \times (B * 5 + C)$ tensor.

```
config.py > ...
S = 7        # Divide each image into a SxS grid
B = 2        # Number of bounding boxes to predict
C = 20       # Number of classes in the dataset
```

evaluating YOLO on PASCAL VOC,

S = 7, B = 2, C = 20

(PASCAL VOC has 20 labelled classes)

final prediction : 7 × 7 × 30 tensor

7 x 7 x (2 x 5 + 20)

$$pred_{cell} = [\underbrace{\boxed{c_1, c_2, \ldots, c_{20}}}_{\text{Class probabilities}}, \underbrace{\boxed{p_{c_1}}}_{\text{Box1 confidence score}}, \underbrace{\boxed{x, y, w, h}}_{\text{Box 1}}, \underbrace{\boxed{p_{c_2}}}_{\text{Box2 confidence score}}, \underbrace{\boxed{x, y, w, h}}_{\text{Box 2}}]$$

VISION SYSTEM LABORATORY

2. Unified Detection

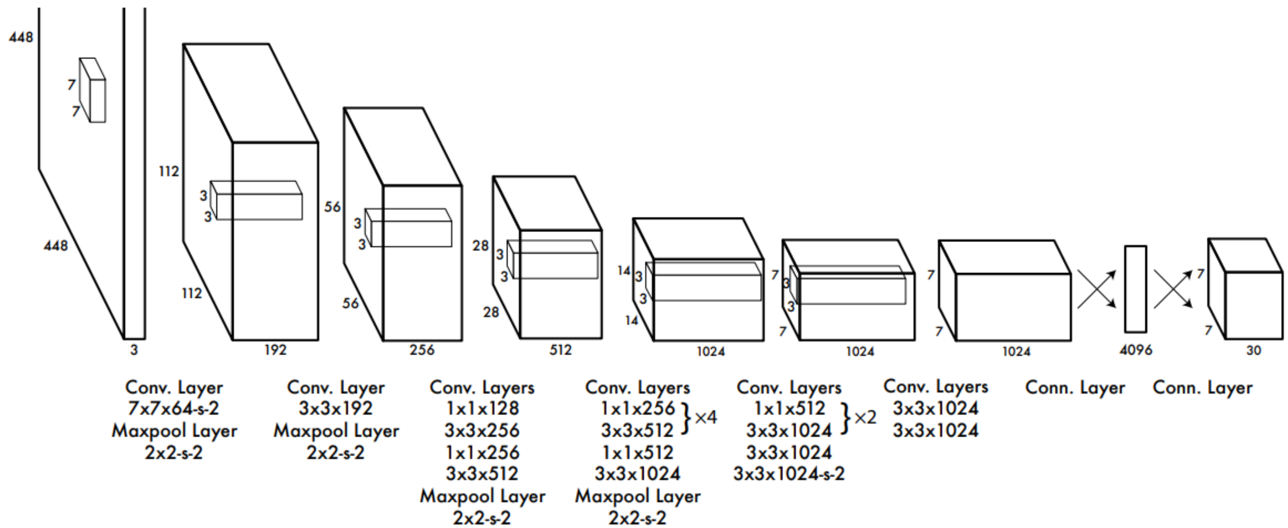


Figure 3: The Architecture. Our detection network has 24 convolutional layers followed by 2 fully connected layers. Alternating 1×1 convolutional layers reduce the features space from preceding layers. We pretrain the convolutional layers on the ImageNet classification task at half the resolution (224×224 input image) and then double the resolution for detection.

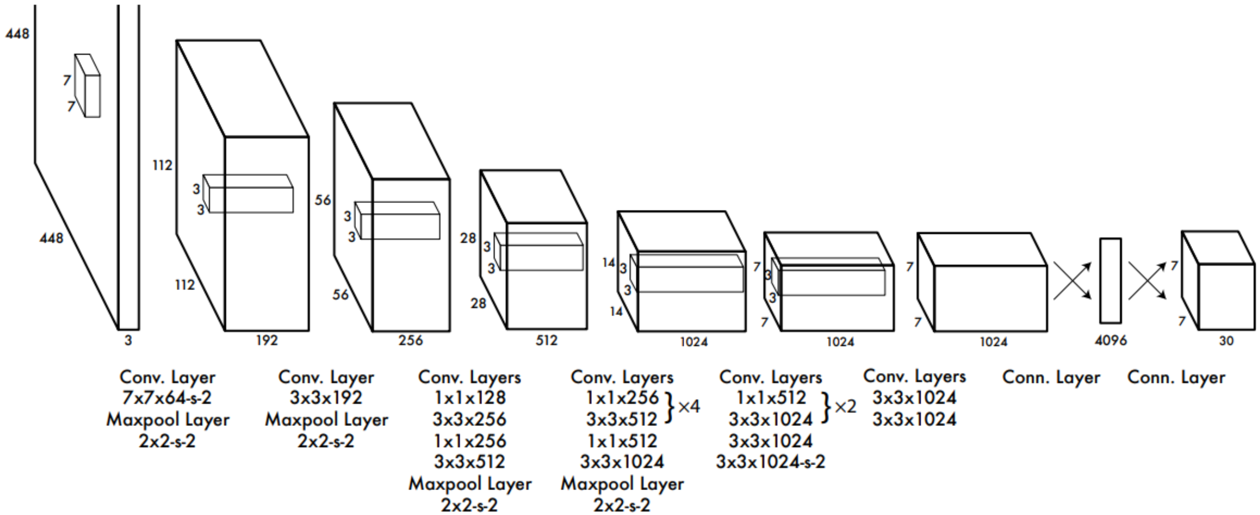
- pretrain convolutional layers on the ImageNet 1000-class competition dataset
- 88% accuracy, comparable to the GoogLeNet models

```
# Load backbone ResNet
backbone = resnet50(weights=ResNet50_Weights.DEFAULT)
backbone.requires_grad_(False)          # Freeze backbone weights
```

Methods

For the sake of convenience, PyTorch's pretrained `ResNet50` architecture was used as the backbone for the model instead of `Darknet`. However, the detection layers at the end of the model exactly follow those described in the

2. Unified Detection




```python
lass YOLOv1(nn.Module):
    def __init__(self):
        super().__init__()
        self.depth = config.B * 5 + config.C

        layers = [
            # Probe(0, forward=lambda x: print('#' * 5 + ' Start ' + '#' * 5)),
            nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3),
            nn.LeakyReLU(negative_slope=0.1),
            # Probe('conv1', forward=probe_dist),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(64, 192, kernel_size=3, padding=1),
            nn.LeakyReLU(negative_slope=0.1),
            # Probe('conv2', forward=probe_dist),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(192, 128, kernel_size=1),
            nn.LeakyReLU(negative_slope=0.1),
            nn.Conv2d(128, 256, kernel_size=3, padding=1),
            nn.LeakyReLU(negative_slope=0.1),
```

final layer

```python
layers += [
    nn.Flatten(),
    nn.Linear(config.S * config.S * 1024, 4096),
    nn.Dropout(),
    nn.LeakyReLU(negative_slope=0.1),
    # Probe('linear1', forward=probe_dist),
    nn.Linear(4096, config.S * config.S * self.depth),
    # Probe('linear2', forward=probe_dist),
]

self.model = nn.Sequential(*layers)
```

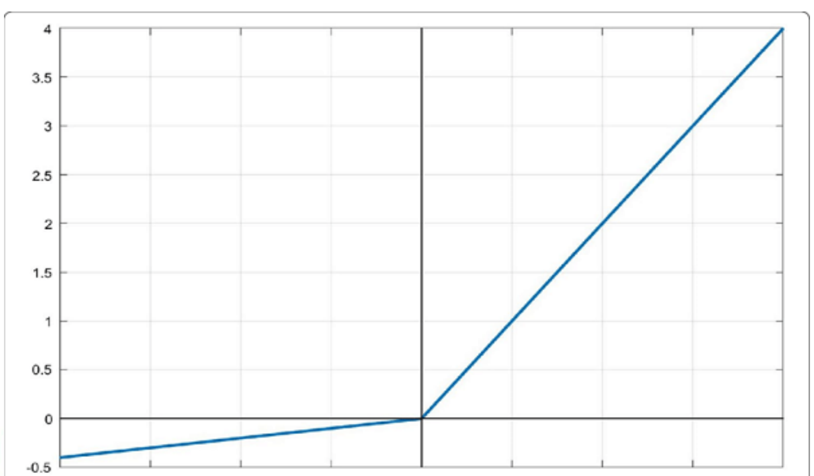
2. Unified Detection

- final layer - linear activation function
- all other layers - leaky rectified linear activation

Leaky ReLU

- small slope when negative
- allows it to convey information while preserving some sort of linearity
- helps the model converge and learn faster

$$\phi(x) = \begin{cases} x, & \text{if } x > 0 \\ 0.1x, & \text{otherwise} \end{cases} \qquad (2)$$




```
class YOLOv1(nn.Module):
    def __init__(self):
        super().__init__()
        self.depth = config.B * 5 + config.C

        layers = [
            # Probe(0, forward=lambda x: print('#'
            nn.Conv2d(3, 64, kernel_size=7, stride=
            nn.LeakyReLU(negative_slope=0.1),
            # Probe('conv1', forward=probe_dist),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(64, 192, kernel_size=3, paddi
            nn.LeakyReLU(negative_slope=0.1),
            # Probe('conv2', forward=probe_dist),
            nn.MaxPool2d(kernel_size=2, stride=2),

            nn.Conv2d(192, 128, kernel_size=1),
            nn.LeakyReLU(negative_slope=0.1),
            nn.Conv2d(128, 256, kernel_size=3, paddi
            nn.LeakyReLU(negative_slope=0.1),
            nn.Conv2d(256, 256, kernel_size=1)
```

final layer

```
layers += [
    nn.Flatten(),
    nn.Linear(config.S * config.S * 1024, 4096),
    nn.Dropout(),
    nn.LeakyReLU(negative_slope=0.1),
    # Probe('linear1', forward=probe_dist),
    nn.Linear(4096, config.S * config.S * self.depth),
    # Probe('linear2', forward=probe_dist),
]
```

Loss Function

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

78

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

79 $$+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(C_i - \hat{C}_i \right)^2$$

80 $$+ \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i \right)^2$$

81 $$+ \sum_{i=0}^{S^2} \mathbb{1}_{i}^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2$$

```
self.l_coord = 5
self.l_noobj = 0.5
```

```
78            total = self.l_coord * (pos_losses + dim_losses) \
79                    + obj_confidence_losses \
80                    + self.l_noobj * noobj_confidence_losses \
81                    + class_losses
82            return total / config.BATCH_SIZE
```

VISION SYSTEM LABORATORY

loss

81

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

- $p_i(c)$: Actual class probabilities
- $\hat{p}_i(c)$: Predicted class probabilities

```
obj_i = bbox_mask[..., 0:1]
```

객체의 존재 여부를 나타내는 마스크,
어떤 객체가 있으면 1이 되는 값
p,a 텐서에 적용- 해당 객체의 클래스에 대한 정보만을 선택

```
        # Classification losses
        class_losses = mse_loss(
            obj_i * p[..., :config.C],
            obj_i * a[..., :config.C]
        )
```

p : prediction
a : ground truth
config.C : 20

0에서 config.C-1까지의 열을 선택

```
def mse_loss(a, b):
    flattened_a = torch.flatten(a, end_dim=-2)
    flattened_b = torch.flatten(b, end_dim=-2).expand_as(flattened_a)
    return F.mse_loss(
        flattened_a,
        flattened_b,
        reduction='sum'
    )
```

loss

79, 80

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

```python
# Confidence losses (target confidence is IOU)
obj_confidence_losses = mse_loss(
    obj_ij * bbox_attr(p, 4),  confidence
    # 객체가 실제로 존재하는 그리드 셀에서 예측된 바운딩 박스의 신뢰도를 가져옴
    obj_ij * torch.ones_like(max_iou)
    # 객체가 존재하는 그리드 셀에서 목표 신뢰도(IoU)를 가져옴
    # torch.ones_like : 목표 신뢰도를 1로 설정
)

noobj_confidence_losses = mse_loss(
    noobj_ij * bbox_attr(p, 4),
    # 객체가 존재하지 않는 그리드 셀에서 예측된 바운딩 박스의 신뢰도를 가져옴
    torch.zeros_like(max_iou)
    # 객체가 존재하지 않는 그리드 셀에서 목표 신뢰도(0)를 가져옴
)
```

```python
def bbox_attr(data, i):  특정 속성에 해당하는 값 추출
    """Returns the Ith attribute of each bounding box in data."
    attr_start = config.C + i
    return data[..., attr_start::5]
```

- $1_{i,j}^{noobj}$: index parameter, 1 when the jth bounding box of the ith grid cell is not responsible to predict an object, and 0 otherwise
- C_i : 1 if the object is included, 0 otherwise.
- $\hat{C}_i$: Confidence score of predicted bounding box

loss

```
# XY position losses
x_losses = mse_loss(
    obj_ij * bbox_attr(p, 0),
    obj_ij * bbox_attr(a, 0)
)
y_losses = mse_loss(
    obj_ij * bbox_attr(p, 1),
    obj_ij * bbox_attr(a, 1)
)
pos_losses = x_losses + y_losses
# print('pos_losses', pos_losses.i
```

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

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

```
# Bbox dimension losses
p_width = bbox_attr(p, 2)
a_width = bbox_attr(a, 2)
width_losses = mse_loss(
    obj_ij * torch.sign(p_width) * torch.sqrt(torch.abs(p_width) + config.EPSILON),
    obj_ij * torch.sqrt(a_width)
)
p_height = bbox_attr(p, 3)
a_height = bbox_attr(a, 3)
height_losses = mse_loss(
    obj_ij * torch.sign(p_height) * torch.sqrt(torch.abs(p_height) + config.EPSILON),
    obj_ij * torch.sqrt(a_height)
)
dim_losses = width_losses + height_losses
# print('dim_losses', dim_losses.item())
```

torch.sign - 부호(양수, 음수, 0) 반환
0으로 나눠지는것 방지 – 작은 상수(config.EPSILON) 더한 후의 제곱근 계산

- S^2 : number of grid cells (7^2 = 49)
- B : number of bounding boxes per grid cell (=2)
- $1_{i,j}^{obj}$: index parameter assigned as 1 if the j-th bounding box in the i-th grid cell is responsible for predicting an object, and 0 otherwise
- x_i, y_i, w_i, h_i : x, y coordinates and width, height of the ground truth box
- $\hat{x}_i, \hat{y}_i, \hat{w}_i, \hat{h}_i$: x, y coordinates, width, height of predicted bounding box

Training

- 135 epochs
- batch size 64
- momentum 0.9
- decay 0.0005

config.py > ...

```python
BATCH_SIZE = 64
EPOCHS = 135
WARMUP_EPOCHS = 0
LEARNING_RATE = 1E-4
```

- For the first epochs - slowly raise the learning rate from 10^−3 to 10^−2
- (start at a high learning rate - model often diverges due to unstable gradients)
- continue training with 10^−2 for 75 epochs, then 10^−3 for 30 epochs, and finally 10^−4 for 30 epochs

```python
# Learning rate scheduler (NOT NEEDED)
# scheduler = torch.optim.lr_scheduler.LambdaLR(
#     optimizer,
#     lr_lambda=utils.scheduler_lambda
# )
```

```python
def scheduler_lambda(epoch):
    if epoch < config.WARMUP_EPOCHS + 75:
        return 1
    elif epoch < config.WARMUP_EPOCHS + 105:
        return 0.1
    else:
        return 0.01
```

VISION SYSTEM LABORATORY

Training

To avoid overfitting - use **dropout** and **extensive data augmentation**

- dropout rate = .5
- nn.Dropout() – default 0.5

```python
class YOLOv1(nn.Module):
```

```python
        layers += [
            nn.Flatten(),
            nn.Linear(config.S * config.S * 1024, 4096),
            nn.Dropout(),
            nn.LeakyReLU(negative_slope=0.1),
            # Probe('linear1', forward=probe_dist),
            nn.Linear(4096, config.S * config.S * self.depth),
            # Probe('linear2', forward=probe_dist),
        ]
```

Training

- random scaling and translations of up to 20% of the original image size
- randomly adjust the exposure and saturation of the image by up to a factor of 1.5 in the HSV color space (paper)
- data was augmented by randomly scaling dimensions, shifting position, and adjusting hue/saturation values by up to 20% of their original values (GitHub ,tanjeffreyz/yolo-v1)

```
# Augment images
if self.augment:
    data = TF.affine(data, angle=0.0, scale=scale, translate=(x_shift, y_shift), shear=0.0)
```

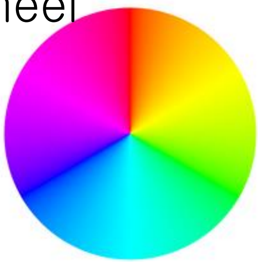
```
x_shift = int((0.2 * random.random() - 0.1) * config.IMAGE_SIZE[0])
y_shift = int((0.2 * random.random() - 0.1) * config.IMAGE_SIZE[1])
scale = 1 + 0.2 * random.random()
```

- angle=0.0: 회전 각도 0, 회전 수행 x
- translate=(x_shift, y_shift): 이미지를 수평 및 수직으로 이동 (up to 20%)

```
    data = TF.adjust_hue(data, 0.2 * random.random() - 0.1)
```

- 색상 조절 (랜덤값[0, 1)에 0.2를 곱하고 0.1을 뺌 -> 범위가 [-0.1, 0.1)으로 조절)
- hue_factor 값이 0.0이면 변화 없음을 의미하며, 양수 값은 시계 방향으로, 음수 값은 반시계 방향으로 색상을 조절 (360도의 color wheel)

color wheel



```
    data = TF.adjust_saturation(data, 0.2 * random.random() + 0.9)
```

- 채도 조절, [0.9, 1.1)범위의 랜덤값 사용

VISION SYSTEM LABORATORY

NMS

- non-maximal suppression adds 2- 3% in mAP

```python
# Non-maximum suppression and render image
image = T.ToPILImage()(data)  # Tensor를 PIL 이미지로 변환
draw = ImageDraw.Draw(image)  # 이미지에 그림을 그리기 위한 객체 생성
discarded = set()  # 제거된 바운딩 박스의 인덱스를 저장하기 위한 set

# Iterate through each bounding box
for i in range(num_boxes):
    if i not in discarded:  # 제거되지 않은 바운딩 박스에 대해서만 수행
        tl, width, height, confidence, class_index = bboxes[i]

        # Decrease confidence of other conflicting bboxes
        for j in range(num_boxes):
            other_class = bboxes[j][4] # j번째 바운딩 박스의 class_index

            # If the other box has the same class and high IoU, decrease its confidence
            if j != i and other_class == class_index and iou[i][j] > max_overlap:
                discarded.add(j)  # IoU가 큰 다른 바운딩 박스를 제거

        # Annotate image
        draw.rectangle((tl, (tl[0] + width, tl[1] + height)), outline='orange')
        # 바운딩 박스를 주황색으로 그림
        text_pos = (max(0, tl[0]), max(0, tl[1] - 11))  # 텍스트 위치 계산
        text = f'{classes[class_index]} {round(confidence * 100, 1)}%' # 클래스와 신뢰도 정보
        text_bbox = draw.textbbox(text_pos, text) # 텍스트 영역 계산
        draw.rectangle(text_bbox, fill='orange')  # 텍스트 배경을 주황색으로 채움
        draw.text(text_pos, text)  # 텍스트 그리기
```

VISION SYSTEM LABORATORY

Limitations of YOLO

Spatial constraints

- each grid cell only predicts two boxes and can only have one class
- limits the number of nearby objects
- struggles with small objects that appear in groups, such as flocks of birds

Difficulty in Generalization

- struggles to generalize to objects in new or unusual aspect ratios or configurations

Limitations of the Loss Function

- loss function treats errors the same in small bounding boxes versus large bounding boxes
- a small error in a small box has a much greater effect on IOU
- incorrect localizations

VISION SYSTEM LABORATORY

4.1. Comparison to Other Real-Time Systems

Real-Time Detectors	Train	mAP	FPS
100Hz DPM [31]	2007	16.0	100
30Hz DPM [31]	2007	26.1	30
Fast YOLO	2007+2012	52.7	**155**
YOLO	2007+2012	**63.4**	45
Less Than Real-Time			
Fastest DPM [38]	2007	30.4	15
R-CNN Minus R [20]	2007	53.5	6
Fast R-CNN [14]	2007+2012	70.0	0.5
Faster R-CNN VGG-16[28]	2007+2012	73.2	7
Faster R-CNN ZF [28]	2007+2012	62.1	18
YOLO VGG-16	2007+2012	66.4	21

Table 1: Real-Time Systems on PASCAL VOC 2007. Comparing the performance and speed of fast detectors. Fast YOLO is the fastest detector on record for PASCAL VOC detection and is still twice as accurate as any other real-time detector. YOLO is 10 mAP more accurate than the fast version while still well above real-time in speed.

- Fast YOLO - the fastest object detection method
- YOLO - 63.4% mAP, real-time performance
- train YOLO using VGG-16 : more accurate, significantly slower than YOLO

VISION SYSTEM LABORATORY

4.2. VOC 2007 Error Analysis

Fast R-CNN is one of the highest performing detectors on PASCAL and it's detections are publicly available

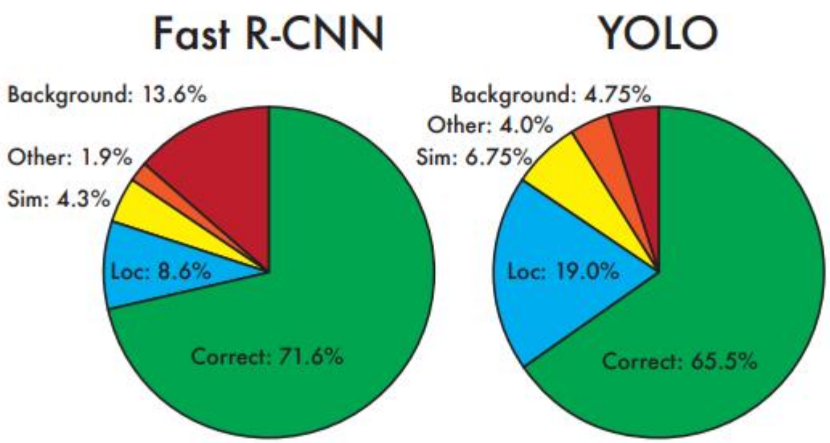


Figure 4: Error Analysis: Fast R-CNN vs. YOLO These charts show the percentage of localization and background errors in the top N detections for various categories (N = # objects in that category).

- Correct: correct class and IOU > .5
- Localization: correct class, .1 < IOU < .5
- Similar: class is similar, IOU > .1
- Other: class is wrong, IOU > .1
- Background: IOU < .1 for any object

VISION SYSTEM LABORATORY

4.3. Combining Fast R-CNN and YOLO

	mAP	Combined	Gain
Fast R-CNN	71.8	-	-
Fast R-CNN (2007 data)	**66.9**	72.4	.6
Fast R-CNN (VGG-M)	59.2	72.4	.6
Fast R-CNN (CaffeNet)	57.1	72.1	.3
YOLO	63.4	**75.0**	**3.2**

Table 2: Model combination experiments on VOC 2007. We examine the effect of combining various models with the best version of Fast R-CNN. Other versions of Fast R-CNN provide only a small benefit while YOLO provides a significant performance boost.

- By using YOLO to eliminate background detections from Fast R-CNN we get a significant boost in performance

- doesn't benefit from the speed of YOLO

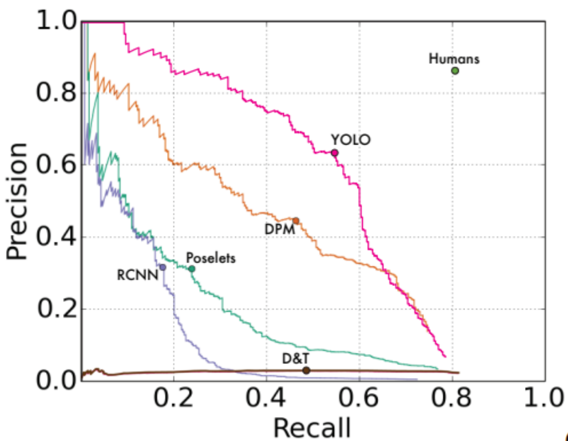
4.4. VOC 2012 Results

VOC 2012 test	mAP	aero	bike	bird	boat	bottle	bus	car	cat	chair	cow	table	dog	horse	mbike	person	plant	sheep	sofa	train	tv
MR_CNN_MORE_DATA [11]	**73.9**	**85.5**	**82.9**	**76.6**	**57.8**	**62.7**	**79.4**	77.2	86.6	**55.0**	**79.1**	**62.2**	87.0	**83.4**	**84.7**	78.9	45.3	73.4	65.8	80.3	74.0
HyperNet_VGG	71.4	84.2	78.5	73.6	55.6	53.7	78.7	**79.8**	87.7	49.6	74.9	52.1	86.0	81.7	83.3	**81.8**	**48.6**	**73.5**	59.4	79.9	65.7
HyperNet_SP	71.3	84.1	78.3	73.3	55.5	53.6	78.6	79.6	87.5	49.5	74.9	52.1	85.6	81.6	83.2	81.6	48.4	73.2	59.3	79.7	65.6
Fast R-CNN + YOLO	70.7	83.4	78.5	73.5	55.8	43.4	79.1	73.1	**89.4**	49.4	75.5	57.0	**87.5**	80.9	81.0	74.7	41.8	71.5	68.5	**82.1**	67.2
MR_CNN_S_CNN [11]	70.7	85.0	79.6	71.5	55.3	57.7	76.0	73.9	84.6	50.5	74.3	61.7	85.5	79.9	81.7	76.4	41.0	69.0	61.2	77.7	72.1
Faster R-CNN [28]	70.4	84.9	79.8	74.3	53.9	49.8	77.5	75.9	88.5	45.6	77.1	55.3	86.9	81.7	80.9	79.6	40.1	72.6	60.9	81.2	61.5
DEEP_ENS_COCO	70.1	84.0	79.4	71.6	51.9	51.1	74.1	72.1	88.6	48.3	73.4	57.8	86.1	80.0	80.7	70.4	46.6	69.6	**68.8**	75.9	71.4
NoC [29]	68.8	82.8	79.0	71.6	52.3	53.7	74.1	69.0	84.9	46.9	74.3	53.1	85.0	81.3	79.5	72.2	38.9	72.4	59.5	76.7	68.1
Fast R-CNN [14]	68.4	82.3	78.4	70.8	52.3	38.7	77.8	71.6	89.3	44.2	73.0	55.0	**87.5**	80.5	80.8	72.0	35.1	68.3	65.7	80.4	64.2
UMICH_FGS_STRUCT	66.4	82.9	76.1	64.1	44.6	49.4	70.3	71.2	84.6	42.7	68.6	55.8	82.7	77.1	79.9	68.7	41.4	69.0	60.0	72.0	66.2
NUS_NIN_C2000 [7]	63.8	80.2	73.8	61.9	43.7	43.0	70.3	67.6	80.7	41.9	69.7	51.7	78.2	75.2	76.9	65.1	38.6	68.3	58.0	68.7	63.3
BabyLearning [7]	63.2	78.0	74.2	61.3	45.7	42.7	68.2	66.8	80.2	40.6	70.0	49.8	79.0	74.5	77.9	64.0	35.3	67.9	55.7	68.7	62.6
NUS_NIN	62.4	77.9	73.1	62.6	39.5	43.3	69.1	66.4	78.9	39.1	68.1	50.0	77.2	71.3	76.1	64.7	38.4	66.9	56.2	66.9	62.7
R-CNN VGG BB [13]	62.4	79.6	72.7	61.9	41.2	41.9	65.9	66.4	84.6	38.5	67.2	46.7	82.0	74.8	76.0	65.2	35.6	65.4	54.2	67.4	60.3
R-CNN VGG [13]	59.2	76.8	70.9	56.6	37.5	36.9	62.9	63.6	81.1	35.7	64.3	43.9	80.4	71.6	74.0	60.0	30.8	63.4	52.0	63.5	58.7
YOLO	57.9	77.0	67.2	57.7	38.3	22.7	68.3	55.9	81.4	36.2	60.8	48.5	77.2	72.3	71.3	63.5	28.9	52.2	54.8	73.9	50.8
Feature Edit [33]	56.3	74.6	69.1	54.4	39.1	33.1	65.2	62.7	69.7	30.8	56.0	44.6	70.0	64.4	71.1	60.2	33.3	61.3	46.4	61.7	57.8
R-CNN BB [13]	53.3	71.8	65.8	52.0	34.1	32.6	59.6	60.0	69.8	27.6	52.0	41.7	69.6	61.3	68.3	57.8	29.6	57.8	40.9	59.3	54.1
SDS [16]	50.7	69.7	58.4	48.5	28.3	28.8	61.3	57.5	70.8	24.1	50.7	35.9	64.9	59.1	65.8	57.1	26.0	58.8	38.6	58.9	50.7
R-CNN [13]	49.6	68.1	63.8	46.1	29.4	27.9	56.6	57.0	65.9	26.5	48.7	39.5	66.2	57.3	65.4	53.2	26.2	54.5	38.1	50.6	51.6

Table 3: PASCAL VOC 2012 Leaderboard. YOLO compared with the full `comp4` (outside data allowed) public leaderboard as of November 6th, 2015. Mean average precision and per-class average precision are shown for a variety of detection methods. YOLO is the only real-time detector. Fast R-CNN + YOLO is the forth highest scoring method, with a 2.3% boost over Fast R-CNN.

- struggles with small objects compared
- categories like bottle, sheep, and tv/monitor YOLO scores 8-10% lower than R-CNN or Feature Edit
- categories like cat and train YOLO achieves higher performance
- combined Fast R-CNN + YOLO model is one of the highest performing detection methods

4.5. Generalizability: Person Detection in Artwork



(a) Picasso Dataset precision-recall curves.

	VOC 2007	Picasso		People-Art
	AP	AP	Best F_1	AP
YOLO	**59.2**	**53.3**	**0.590**	**45**
R-CNN	54.2	10.4	0.226	26
DPM	43.2	37.8	0.458	32
Poselets [2]	36.5	17.8	0.271	
D&T [4]	-	1.9	0.051	

(b) Quantitative results on the VOC 2007, Picasso, and People-Art Datasets. The Picasso Dataset evaluates on both AP and best F_1 score.

Figure 5: Generalization results on Picasso and People-Art datasets.

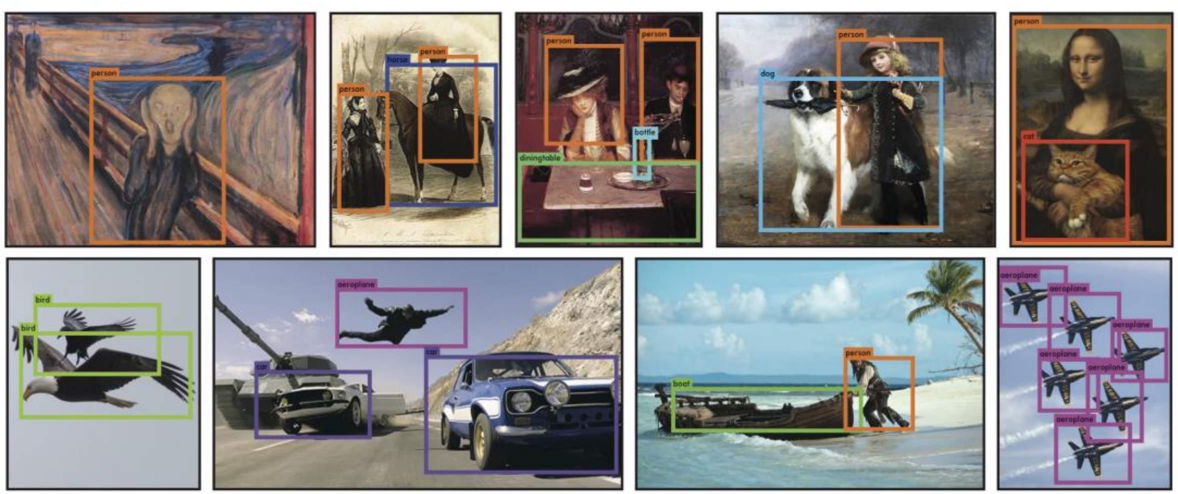


Figure 6: Qualitative Results. YOLO running on sample artwork and natural images from the internet. It is mostly accurate although it does think one person is an airplane.

testing person detection on artwork

R-CNN drops off considerably when applied to artwork

DPM maintains its AP well when applied to artwork

YOLO has good performance

YOLO models the size and shape of objects, as well as relationships between objects and where objects commonly appear

Artwork and natural images are similar in terms of the size and shape of objects, thus YOLO can still predict good bounding boxes and detections

6. Conclusion

YOLO, a unified model for object detection

- simple to construct
- trained on a loss function that directly corresponds to detection performance

- Fast YOLO is the fastest general-purpose object detector in the literature and YOLO pushes the state-of-the-art in real-time object detection
- YOLO generalizes well to new domains making it ideal for applications that rely on fast, robust object detection