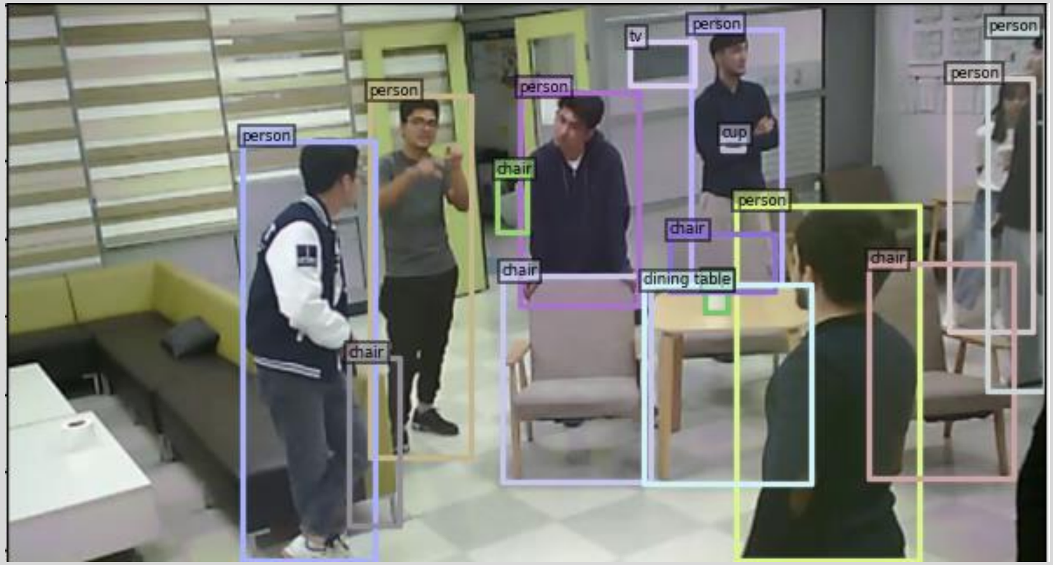
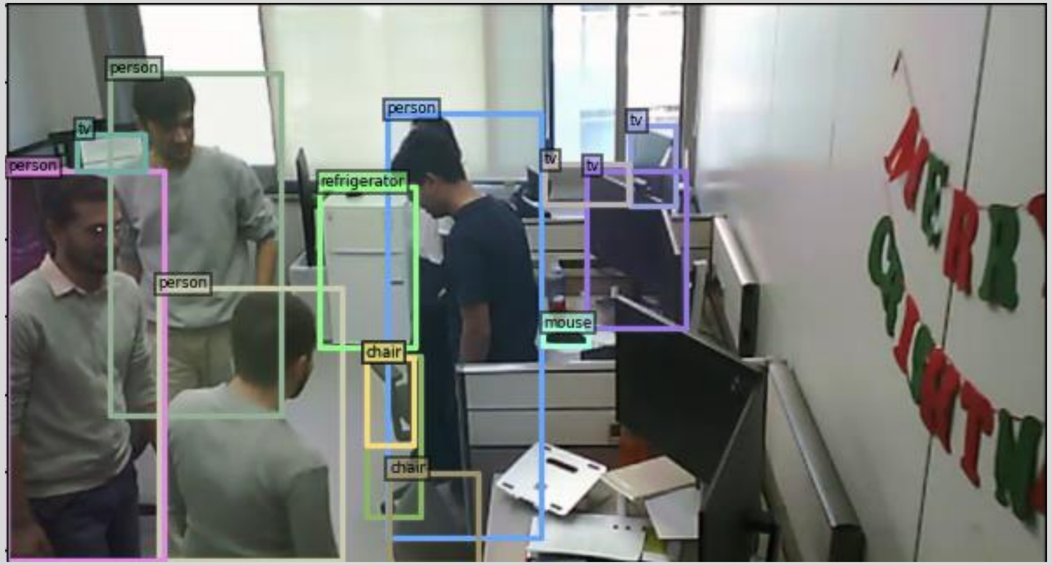
Computer Vision

Lecture 07: Real-time object detection pipeline

Final project





Real-time human box detection task

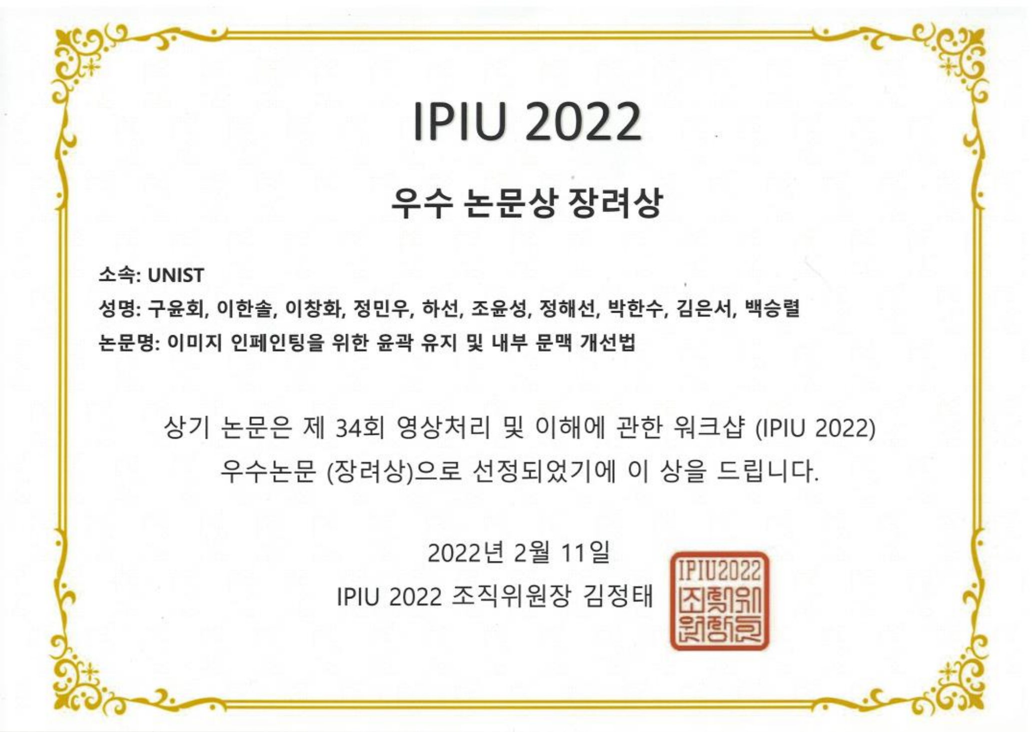
Final project

Will be noticed soon:

Mid-Presentation Video Submission: 11th Nov. 23:29
Mid-Presentation Video Watching + Q&A: 13th/15th Nov. Classes
Final Presentation Video Submission: 1st Dec. 23:29
Final Presentation video Watching + Q&A: 4th/6th Dec. Classes
Final Code+Data+Report Submission : 22th Dec. 23:29

Final project

1st/2nd/3rd prizes: X million won. Evaluated based on the leaderboard.



IPIU 2022

우수 논문상 장려상

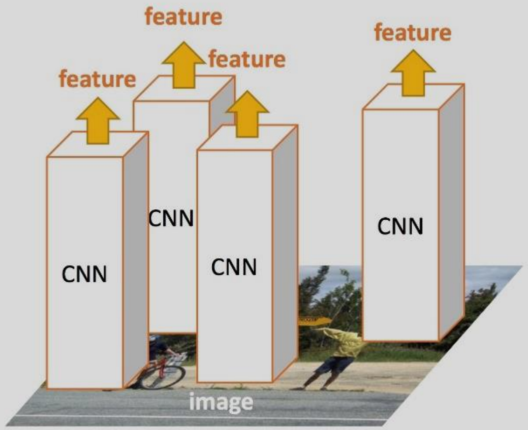
소속: UNIST
성명: 구윤회, 이한솔, 이창화, 정민우, 하선, 조윤성, 정해선, 박한수, 김은서, 백승렬
논문명: 이미지 인페인팅을 위한 윤곽 유지 및 내부 문맥 개선법

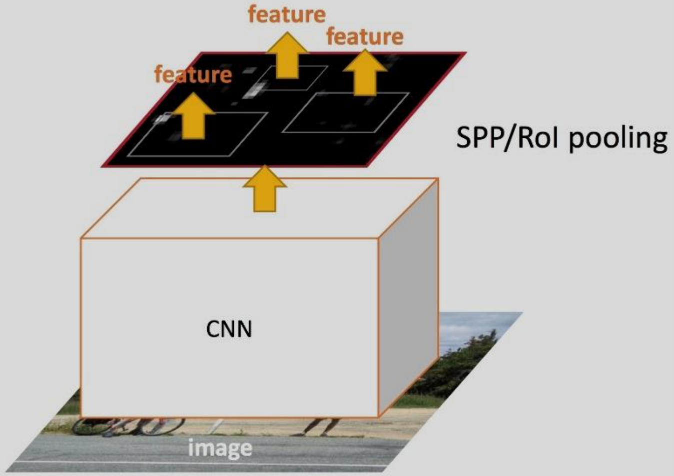
상기 논문은 제 34회 영상처리 및 이해에 관한 워크샵 (IPIU 2022)
우수논문 (장려상)으로 선정되었기에 이 상을 드립니다.

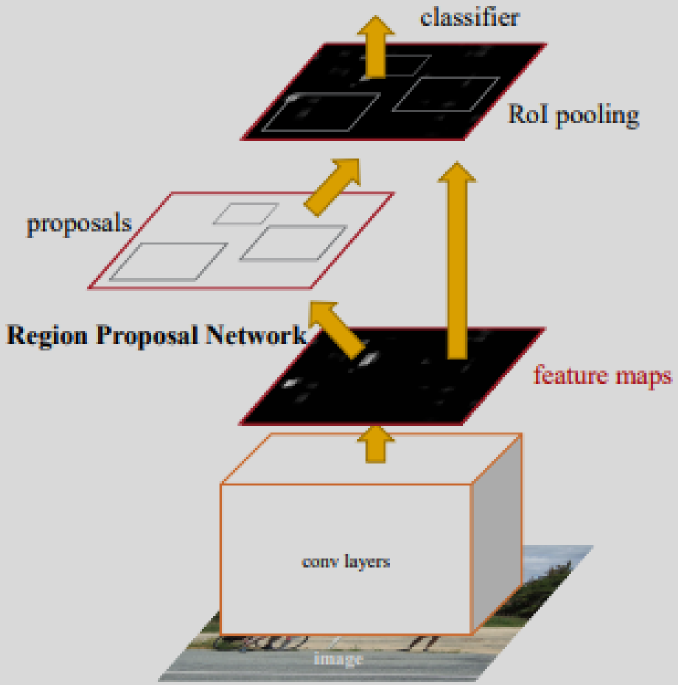
2022년 2월 11일
IPIU 2022 조직위원장 김정태

We will also summarize and submit the results upon approval.

Two-stage method







R-CNN
- Extract image regions
- 1 CNN per region (2000 CNNs)
- Classify region-based features

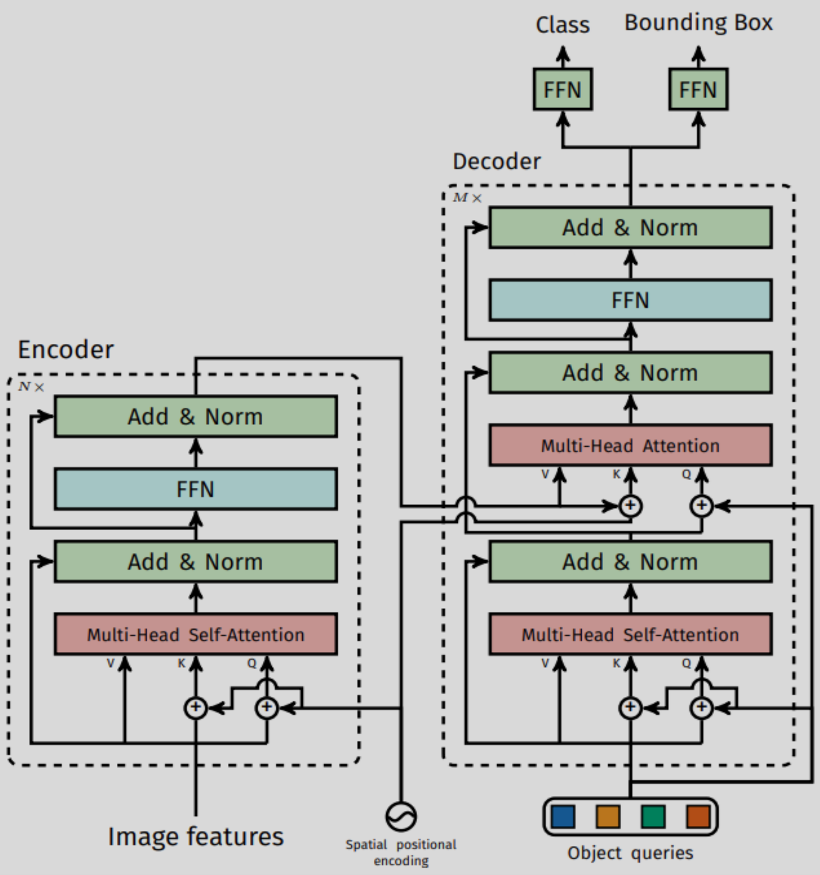
SPP-net & Fast R-CNN (the same forward pipeline)
- 1 CNN on the entire image
- Extract features from feature map regions
- Classify region-based features

System	Time	07 data	07 + 12 data
R-CNN	~ 50s	66.0	-
Fast R-CNN	~ 2s	66.9	70.0
Faster R-CNN	**~ 198ms**	**69.9**	**73.2**

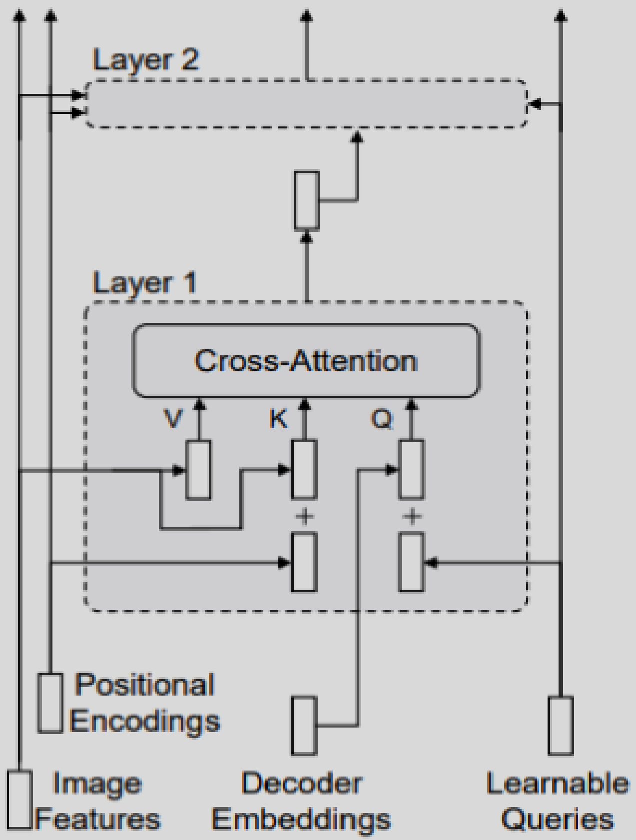
Detection mAP on PASCAL VOC 2007 and 2012, with VGG-16 pre-trained on ImageNet Dataset

DETR-based method

DETR Transformer



Decoder Structure

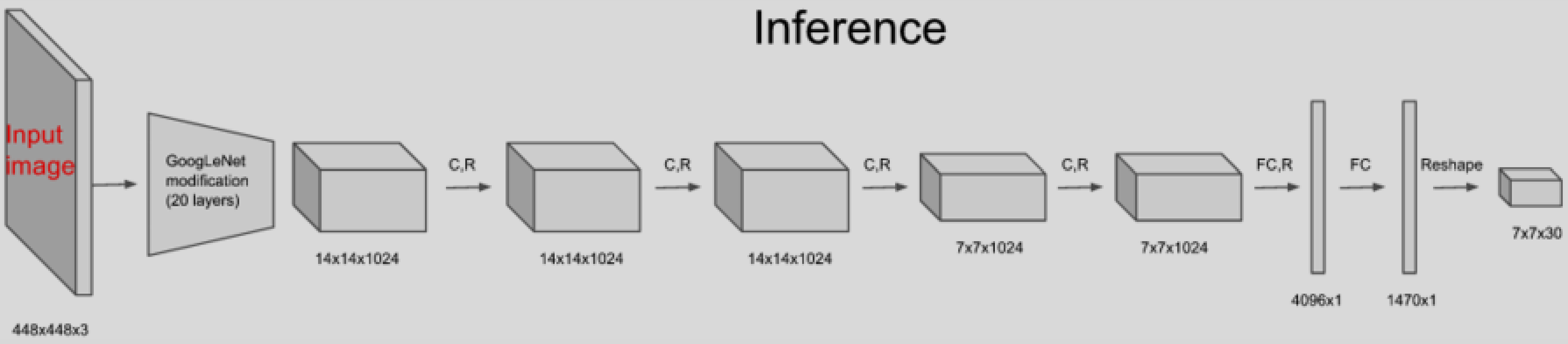


YOLO (You only look once, CVPR'16)

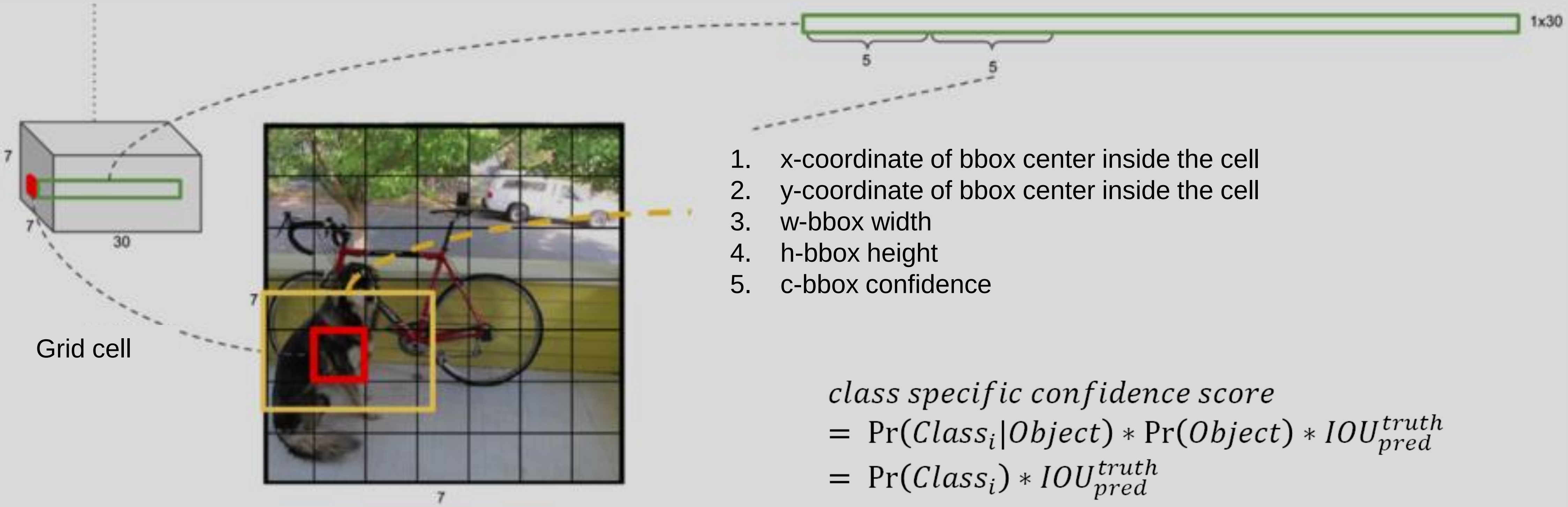
- 1 stage algorithm

YOLO (You only look once, CVPR'16)

- Two candidates for each grid, 20 classes:



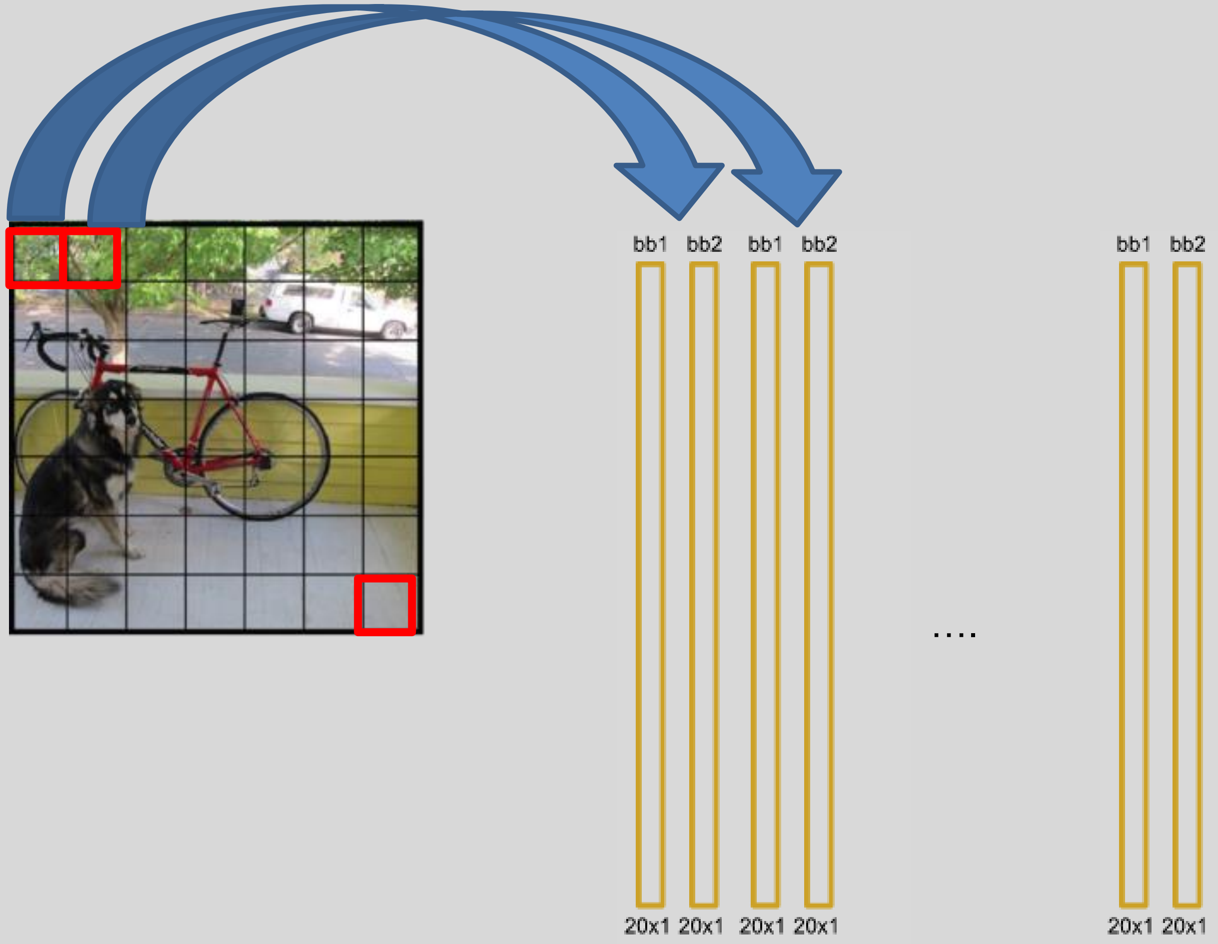
YOLO (You only look once, CVPR'16)



1. x-coordinate of bbox center inside the cell
2. y-coordinate of bbox center inside the cell
3. w-bbox width
4. h-bbox height
5. c-bbox confidence

$$\begin{aligned}&class\ specific\ confidence\ score\\ &= \Pr(Class_i|Object) * \Pr(Object) * IOU_{pred}^{truth}\\ &= \Pr(Class_i) * IOU_{pred}^{truth}\end{aligned}$$

YOLO (You only look once, CVPR’16)



7x7x2=98 boxes

Loss

$$
\lambda_{\textbf{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right]
$$
$$
+ \lambda_{\textbf{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right]
$$
$$
+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(C_i - \hat{C}_i \right)^2
$$
$$
+ \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i \right)^2
$$
$$
+ \sum_{i=0}^{S^2} \mathbb{1}_{i}^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2
$$

Code for generating GTs

```
def encoder(self,boxes,labels):
        '''
        boxes (tensor) [[x1,y1,x2,y2],[]]
        labels (tensor) [...]
        return 7x7x30
        '''
        grid_num = 7
        target = torch.zeros((grid_num,grid_num,30))
        cell_size = 1./grid_num
        wh = boxes[:,2:]-boxes[:,:2]
        cxcy = (boxes[:,2:]+boxes[:,:2])/2
        for i in range(cxcy.size()[0]):
            cxcy_sample = cxcy[i]
            ij = (cxcy_sample/cell_size).ceil()-1
            target[int(ij[1]),int(ij[0]),4] = 1
            target[int(ij[1]),int(ij[0]),9] = 1
            target[int(ij[1]),int(ij[0]),int(labels[i])+9] = 1
            xy = ij*cell_size
            delta_xy = (cxcy_sample -xy)/cell_size
            target[int(ij[1]),int(ij[0]),2:4] = wh[i]
            target[int(ij[1]),int(ij[0]),:2] = delta_xy
            target[int(ij[1]),int(ij[0]),7:9] = wh[i]
            target[int(ij[1]),int(ij[0]),5:7] = delta_xy
        return
```

Loss

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable


class Loss(nn.Module):

    def __init__(self, feature_size=7, num_bboxes=2, num_classes=20, lambda_coord=5.0, lambda_noobj=0.5):
        """ Constructor.
        Args:
            feature_size: (int) size of input feature map (grid).
            num_bboxes: (int) number of bboxes per each cell.
            num_classes: (int) number of the object classes.
            lambda_coord: (float) weight for bbox location/size losses.
            lambda_noobj: (float) weight for no-objectness loss.
        """
        super(Loss, self).__init__()

        self.S = feature_size
        self.B = num_bboxes
        self.C = num_classes
        self.lambda_coord = lambda_coord
        self.lambda_noobj = lambda_noobj
```

Loss

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable


class Loss(nn.Module):

    def forward(self, pred_tensor, target_tensor):
        """ Compute loss for YOLO training.
        Args:
            pred_tensor: (Tensor) predictions, sized [n_batch, S, S, Bx5+C], 5=len([x, y, w, h, conf]).
            target_tensor: (Tensor) targets, sized [n_batch, S, S, Bx5+C].
        Returns:
            (Tensor): loss, sized [1, ].
        """
        # TODO: Romove redundant dimensions for some Tensors.

        S, B, C = self.S, self.B, self.C
        N = 5 * B + C    # 5=len([x, y, w, h, conf]
```

Loss

```
batch_size = pred_tensor.size(0)
coord_mask = target_tensor[:, :, :, 4] > 0
# mask for the cells which contain objects. [n_batch, S, S]
noobj_mask = target_tensor[:, :, :, 4] == 0
# mask for the cells which do not contain objects. [n_batch, S, S]
coord_mask = coord_mask.unsqueeze(-1).expand_as(target_tensor)
# [n_batch, S, S] -> [n_batch, S, S, N]
noobj_mask = noobj_mask.unsqueeze(-1).expand_as(target_tensor)
# [n_batch, S, S] -> [n_batch, S, S, N]

coord_pred = pred_tensor[coord_mask].view(-1, N)
# pred tensor on the cells which contain objects. [n_coord, N]
# n_coord: number of the cells which contain objects.
bbox_pred = coord_pred[:, :5*B].contiguous().view(-1, 5)
# [n_coord x B, 5=len([x, y, w, h, conf])]
class_pred = coord_pred[:, 5*B:]
# [n_coord, C]
coord_target = target_tensor[coord_mask].view(-1, N)
# target tensor on the cells which contain objects. [n_coord, N]
# n_coord: number of the cells which contain objects.
bbox_target = coord_target[:, :5*B].contiguous().view(-1, 5)
# [n_coord x B, 5=len([x, y, w, h, conf])]
class_target = coord_target[:, 5*B:]
# [n_coord, C]
```

Loss

```
# Compute loss for the cells with no object bbox.
noobj_pred = pred_tensor[noobj_mask].view(-1, N)
# pred tensor on the cells which do not contain objects. [n_noobj, N]
# n_noobj: number of the cells which do not contain objects.
noobj_target = target_tensor[noobj_mask].view(-1, N)
# target tensor on the cells which do not contain objects. [n_noobj, N]
# n_noobj: number of the cells which do not contain objects.
noobj_conf_mask = torch.cuda.ByteTensor(noobj_pred.size()).fill_(0) # [n_noobj, N]

for b in range(B):
    noobj_conf_mask[:, 4 + b*5] = 1 # noobj_conf_mask[:, 4] = 1; noobj_conf_mask[:, 9] = 1
noobj_pred_conf = noobj_pred[noobj_conf_mask]       # [n_noobj, 2=len([conf1, conf2])]
noobj_target_conf = noobj_target[noobj_conf_mask]   # [n_noobj, 2=len([conf1, conf2])]
loss_noobj = F.mse_loss(noobj_pred_conf, noobj_target_conf, reduction='sum')

# Compute loss for the cells with objects.
coord_response_mask = torch.cuda.ByteTensor(bbox_target.size()).fill_(0)    # [n_coord x B, 5]
coord_not_response_mask = torch.cuda.ByteTensor(bbox_target.size()).fill_(1)# [n_coord x B, 5]
bbox_target_iou = torch.zeros(bbox_target.size()).cuda()
# [n_coord x B, 5], only the last 1=(conf,) is used
```

Loss

```
        # Choose the predicted bbox having the highest IoU for each target bbox.
        for i in range(0, bbox_target.size(0), B):
            pred = bbox_pred[i:i+B] # predicted bboxes at i-th cell, [B, 5=len([x, y, w, h, conf])]
            pred_xyxy = Variable(torch.FloatTensor(pred.size())) # [B, 5=len([x1, y1, x2, y2, conf])]
            # Because (center_x,center_y)=pred[:, 2] and (w,h)=pred[:,2:4] are normalized for cell-size and image-size respectively,
            # rescale (center_x,center_y) for the image-size to compute IoU correctly.
            pred_xyxy[:,  :2] = pred[:, :2]/float(S) - 0.5 * pred[:, 2:4]
            pred_xyxy[:, 2:4] = pred[:, :2]/float(S) + 0.5 * pred[:, 2:4]

            target = bbox_target[i]
            # target bbox at i-th cell.
            # Because target boxes contained by each cell are identical in current implementation,
            # enough to extract the first one.
            target = bbox_target[i].view(-1, 5) # target bbox at i-th cell, [1, 5=len([x, y, w, h, conf])]
            target_xyxy = Variable(torch.FloatTensor(target.size())) # [1, 5=len([x1, y1, x2, y2, conf])]
            # Because (center_x,center_y)=target[:, 2] and (w,h)=target[:,2:4] are normalized for cell-size and image-size respectively,
            # rescale (center_x,center_y) for the image-size to compute IoU correctly.
            target_xyxy[:,  :2] = target[:, :2]/float(S) - 0.5 * target[:, 2:4]
            target_xyxy[:, 2:4] = target[:, :2]/float(S) + 0.5 * target[:, 2:4]
```

Loss

```
            iou = self.compute_iou(pred_xyxy[:, :4], target_xyxy[:, :4]) # [B, 1]
            max_iou, max_index = iou.max(0)
            max_index = max_index.data.cuda()

            coord_response_mask[i+max_index] = 1
            coord_not_response_mask[i+max_index] = 0

            # "we want the confidence score to equal the intersection over union (IOU) between the predict
ed box and the ground truth"
            # from the original paper of YOLO.
            bbox_target_iou[i+max_index, torch.LongTensor([4]).cuda()] = (max_iou).data.cuda()

        bbox_target_iou = Variable(bbox_target_iou).cuda()

        # BBox location/size and objectness loss for the response bboxes.
        bbox_pred_response = bbox_pred[coord_response_mask].view(-1, 5)      # [n_response, 5]
        bbox_target_response = bbox_target[coord_response_mask].view(-1, 5)
        # [n_response, 5], only the first 4=(x, y, w, h) are used
        target_iou = bbox_target_iou[coord_response_mask].view(-1, 5)
        # [n_response, 5], only the last 1=(conf,) is used
```

Loss

```
        loss_xy = F.mse_loss(bbox_pred_response[:, :2], bbox_target_response[:, :2], reduction='sum')
        loss_wh = F.mse_loss(torch.sqrt(bbox_pred_response[:, 2:4]), torch.sqrt(bbox_target_response[:, 2:
4]), reduction='sum')
        loss_obj = F.mse_loss(bbox_pred_response[:, 4], target_iou[:, 4], reduction='sum')

        # Class probability loss for the cells which contain objects.
        loss_class = F.mse_loss(class_pred, class_target, reduction='sum')

        # Total loss
        loss = self.lambda_coord * (loss_xy + loss_wh) + loss_obj + self.lambda_noobj * loss_noobj + loss_
class
        loss = loss / float(batch_size)

        return loss
```

IoU calculation

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

Loss

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable


class Loss(nn.Module):


    def compute_iou(self, bbox1, bbox2):
        """ Compute the IoU (Intersection over Union) of two set of bboxes, each bbox format: [x1, y1, x2,
 y2].
        Args:
            bbox1: (Tensor) bounding bboxes, sized [N, 4].
            bbox2: (Tensor) bounding bboxes, sized [M, 4].
        Returns:
            (Tensor) IoU, sized [N, M].
        """
        N = bbox1.size(0)
        M = bbox2.size(0)
```

Loss

```
# Compute left-top coordinate of the intersections
lt = torch.max(
    bbox1[:, :2].unsqueeze(1).expand(N, M, 2), # [N, 2] -> [N, 1, 2] -> [N, M, 2]
    bbox2[:, :2].unsqueeze(0).expand(N, M, 2)  # [M, 2] -> [1, M, 2] -> [N, M, 2]
)
# Conpute right-bottom coordinate of the intersections
rb = torch.min(
    bbox1[:, 2:].unsqueeze(1).expand(N, M, 2), # [N, 2] -> [N, 1, 2] -> [N, M, 2]
    bbox2[:, 2:].unsqueeze(0).expand(N, M, 2)  # [M, 2] -> [1, M, 2] -> [N, M, 2]
)
# Compute area of the intersections from the coordinates
wh = rb - lt   # width and height of the intersection, [N, M, 2]
wh[wh < 0] = 0 # clip at 0
inter = wh[:, :, 0] * wh[:, :, 1] # [N, M]

# Compute area of the bboxes
area1 = (bbox1[:, 2] - bbox1[:, 0]) * (bbox1[:, 3] - bbox1[:, 1]) # [N, ]
area2 = (bbox2[:, 2] - bbox2[:, 0]) * (bbox2[:, 3] - bbox2[:, 1]) # [M, ]
area1 = area1.unsqueeze(1).expand_as(inter) # [N, ] -> [N, 1] -> [N, M]
area2 = area2.unsqueeze(0).expand_as(inter) # [M, ] -> [1, M] -> [N, M]

# Compute IoU from the areas
union = area1 + area2 - inter # [N, M, 2]
iou = inter / union           # [N, M, 2]

return iou
```

Non-maximal suppression

Non-maximal suppression

```
def nms(bboxes,scores,threshold=0.5):
    '''
    bboxes(tensor) [N,4]
    scores(tensor) [N,]
    '''
    x1 = bboxes[:,0]
    y1 = bboxes[:,1]
    x2 = bboxes[:,2]
    y2 = bboxes[:,3]

    areas = (x2-x1) * (y2-y1)
    _,order = scores.sort(0,descending=True)
    keep = []
```

Non-maximal suppression

```
while order.numel() > 0:
    i = order[0]
    keep.append(i)

    if order.numel() == 1:
        break

    xx1 = x1[order[1:]].clamp(min=x1[i])
    yy1 = y1[order[1:]].clamp(min=y1[i])
    xx2 = x2[order[1:]].clamp(max=x2[i])
    yy2 = y2[order[1:]].clamp(max=y2[i])

    w = (xx2-xx1).clamp(min=0)
    h = (yy2-yy1).clamp(min=0)
    inter = w*h

    ovr = inter / (areas[i] + areas[order[1:]] - inter)
    ids = (ovr<=threshold).nonzero().squeeze()
    if ids.numel() == 0:
        break
    order = order[ids+1]
return torch.LongTensor(keep)
```

YOLO (You only look once, CVPR'16)

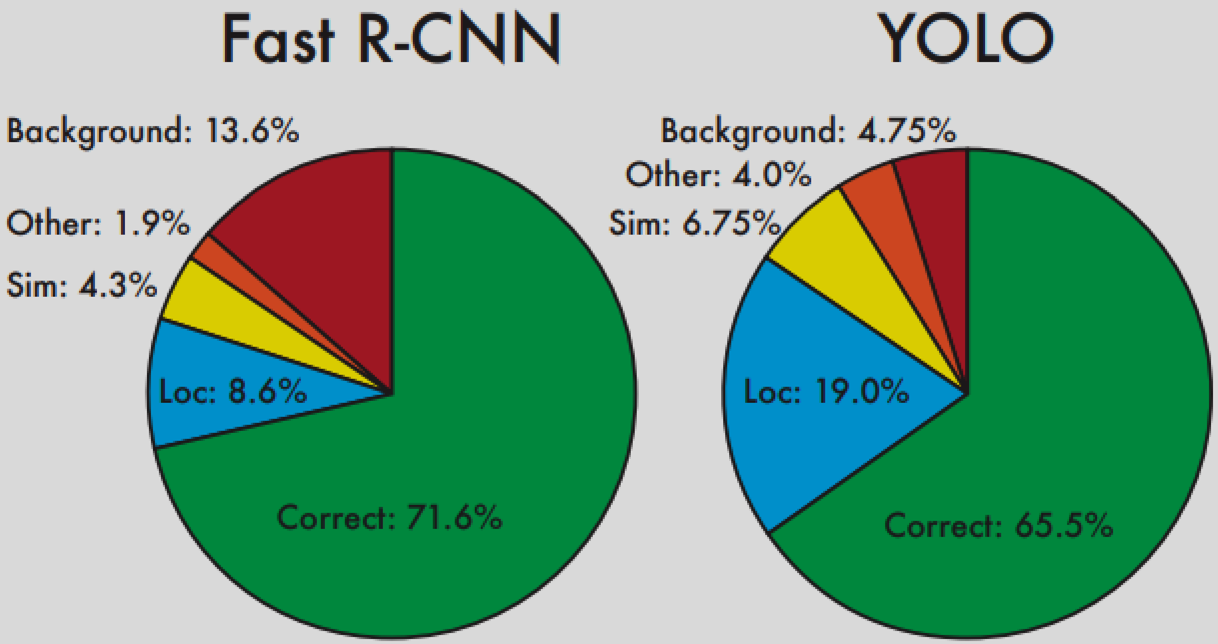


Figure 4: Error Analysis: Fast R-CNN vs. YOLO These charts show the percentage of localization and background errors in the top N detections for various categories (N = # objects in that category).

(1) High localization error
(2) Low recall

Recall vs. Precision

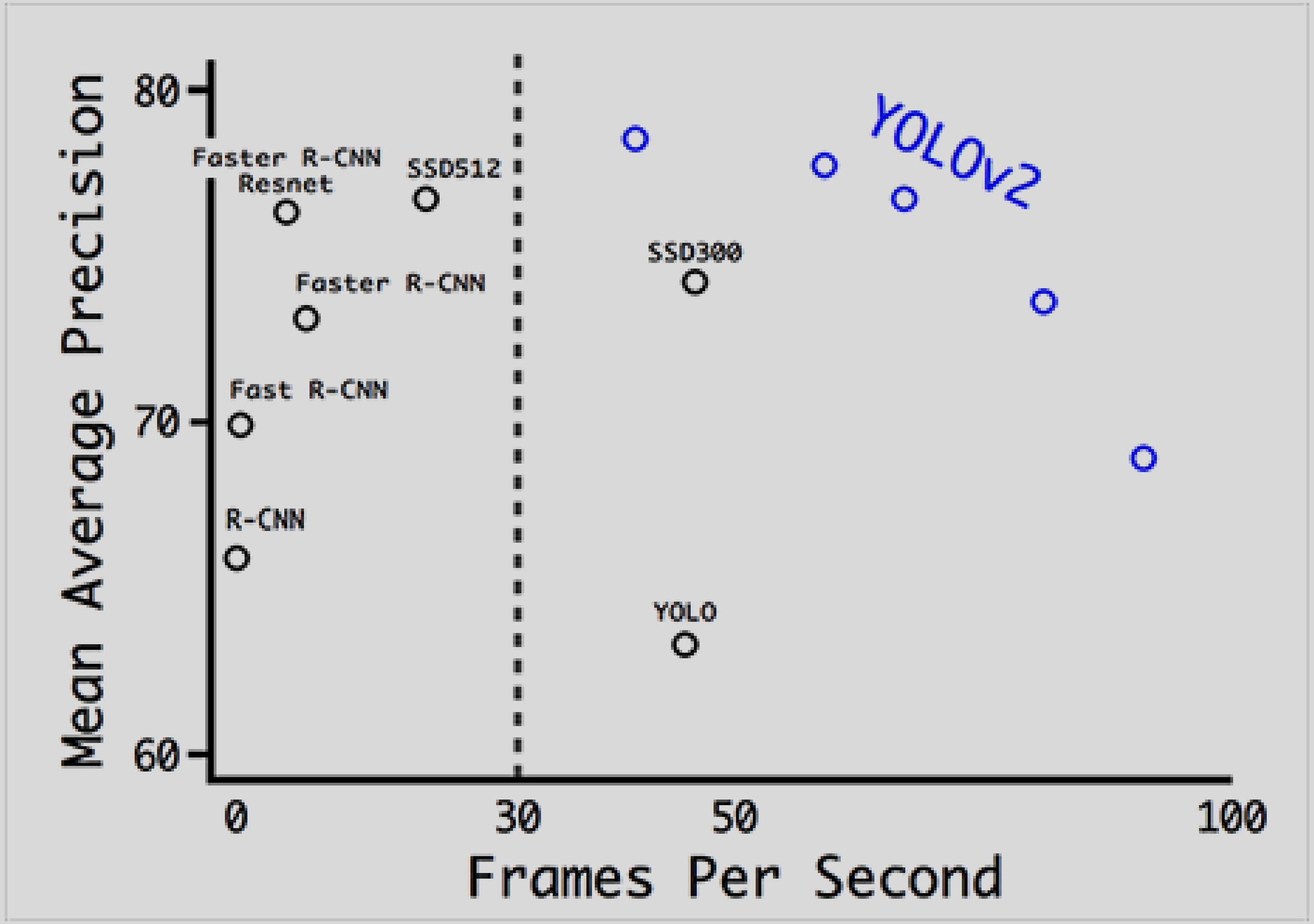
Two measurements for evaluating the performance of the system.
Recall: coverage of the GT, Precision: The correctness ratio among estimations.

Tradeoffs between recall vs. precision.

High recall, may have less precision.

Low recall, may have high precision.

YOLO (You only look once, CVPR’16)

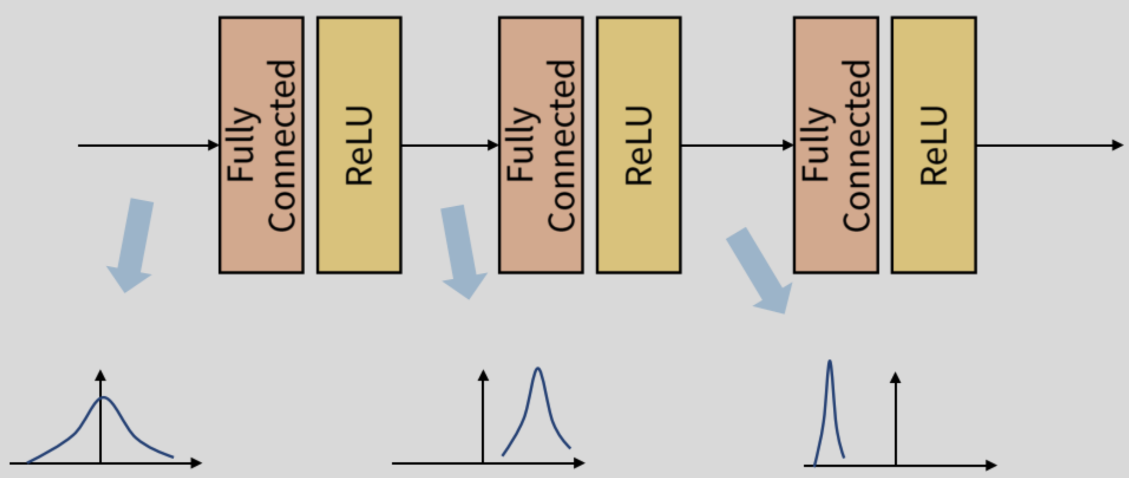


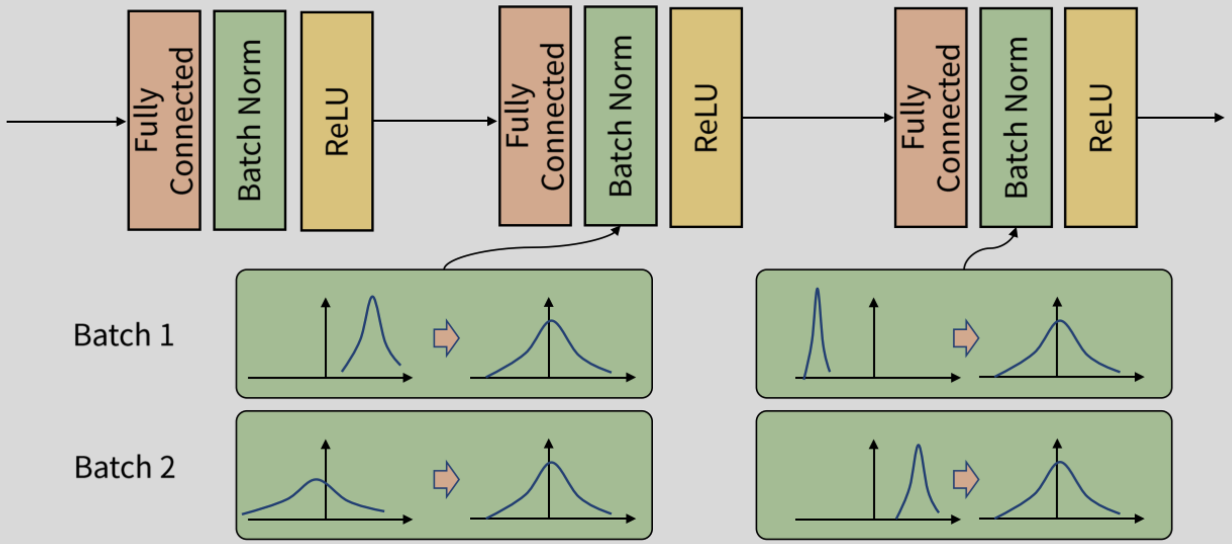
YoloV2

	YOLO								YOLOv2
batch norm?		✓	✓	✓	✓	✓	✓	✓	✓
hi-res classifier?			✓	✓	✓	✓	✓	✓	✓
convolutional?				✓	✓	✓	✓	✓	✓
anchor boxes?				✓	✓				
new network?					✓	✓	✓	✓	✓
dimension priors?						✓	✓	✓	✓
location prediction?						✓	✓	✓	✓
passthrough?							✓	✓	✓
multi-scale?								✓	✓
hi-res detector?									✓
VOC2007 mAP	63.4	65.8	69.5	69.2	69.6	74.4	75.4	76.8	**78.6**

Batch normalization

Internal Covariate Shift:





Changes in the data distribution for each layer.

Applying the batch-normalization.

Batch normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\dots m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

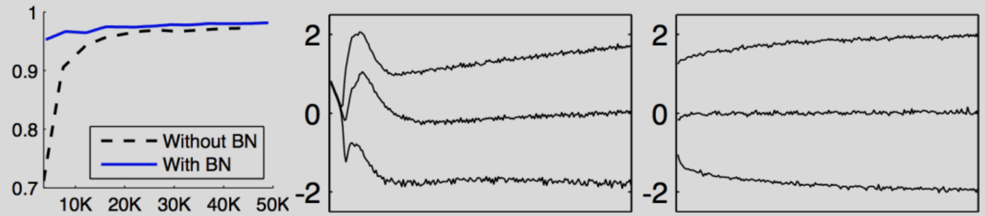
$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.



(a) (b) Without BN (c) With BN

Figure 1: (a) *The test accuracy of the MNIST network trained with and without Batch Normalization, vs. the number of training steps. Batch Normalization helps the network train faster and achieve higher accuracy.* (b, c) *The evolution of input distributions to a typical sigmoid, over the course of training, shown as* $\{15, 50, 85\}$*th percentiles. Batch Normalization makes the distribution more stable and reduces the internal covariate shift.*

Hi-res classifier/detection

- YOLOv1 trains the classifier network (VGG16) at 224x224 and increases the resolution to 448 for detection.
- YOLOv2 fine-tune the classification network at the full 448x448 resolution for 10 epochs on ImageNet.

Hi-res classifier/detection

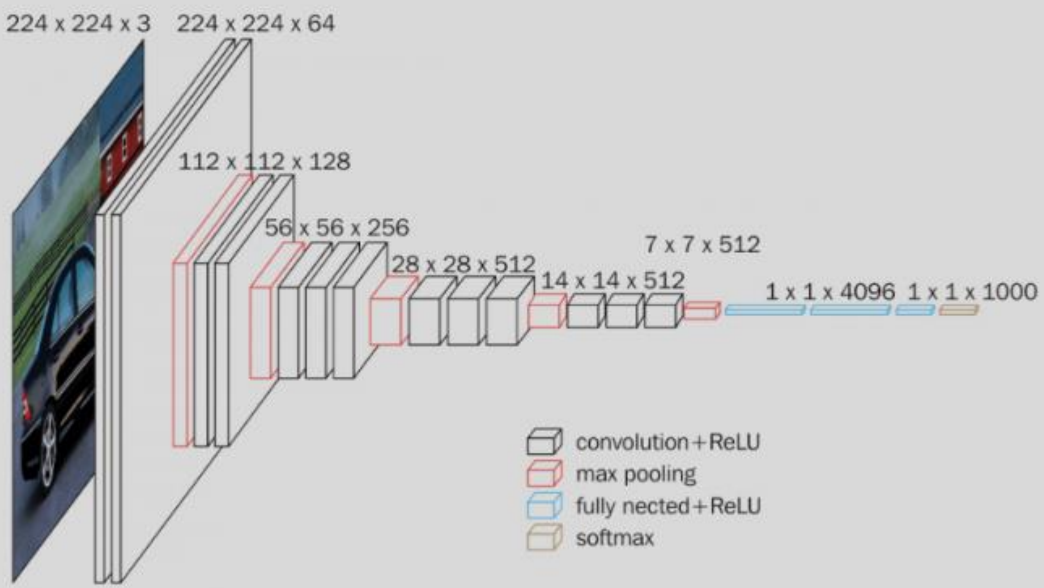
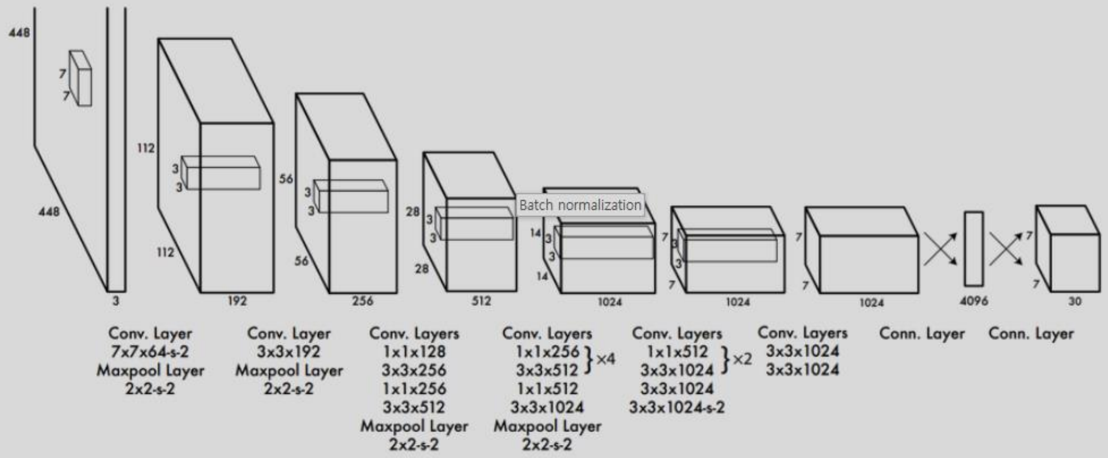


Image classification pre-trained on ImageNet (VGG-16) using 224x224 images.



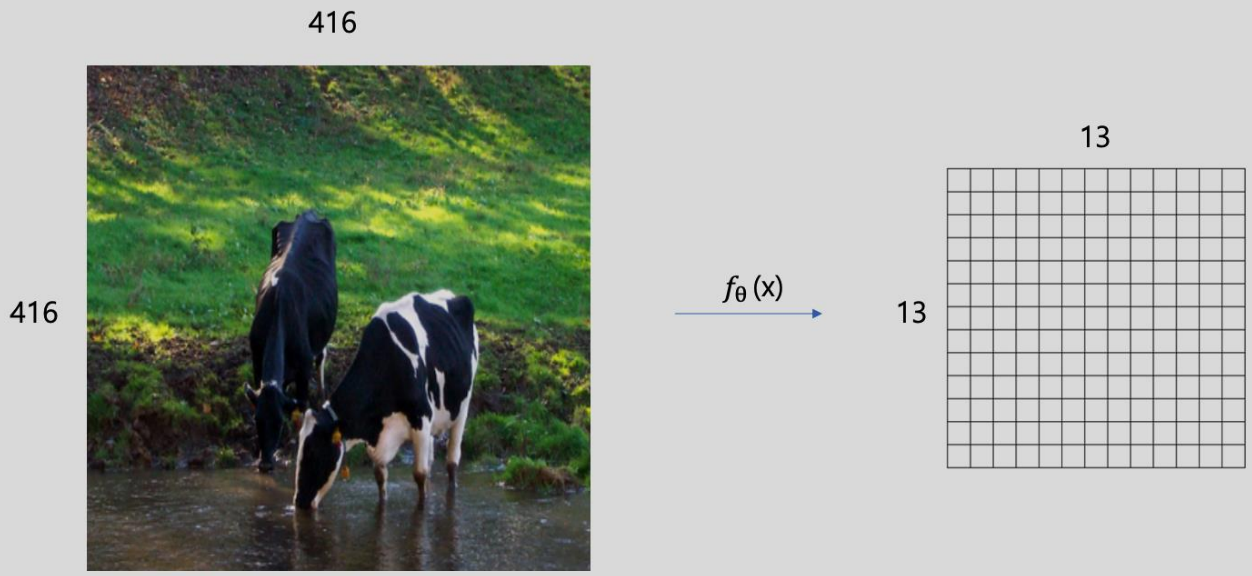
VGG-16 with 448x448 input images for detection.

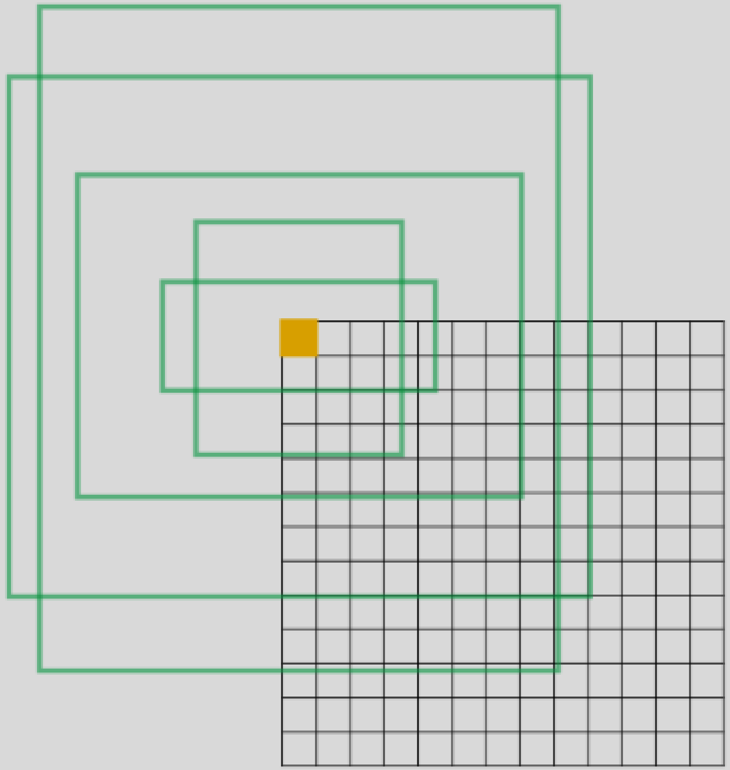
Architectures based on VGG16 for YOLO v1

Convolutional

- Remove FC layers from YOLO.
- Eliminate a pooling layer to make the output of the network’s convolutional layers higher resolution.

Anchor boxes





7x7x2=98 boxes in Yolo v1

13x13x5=845 bounding boxes in Yolo v2

Anchor boxes

	YOLO								YOLOv2
batch norm?		✓	✓	✓	✓	✓	✓	✓	✓
hi-res classifier?			✓	✓	✓	✓	✓	✓	✓
convolutional?				✓	✓	✓	✓	✓	✓
anchor boxes?				✓	✓				
new network?					✓	✓	✓	✓	✓
dimension priors?						✓	✓	✓	✓
location prediction?						✓	✓	✓	✓
passthrough?							✓	✓	✓
multi-scale?								✓	✓
hi-res detector?									✓
VOC2007 mAP	63.4	65.8	69.5	69.2	69.6	74.4	75.4	76.8	**78.6**

mAP becomes less; while it raises the recall from 81% to 88%.

Dimension priors

- Run k-means clustering on training set bounding boxes to automatically find good priors.

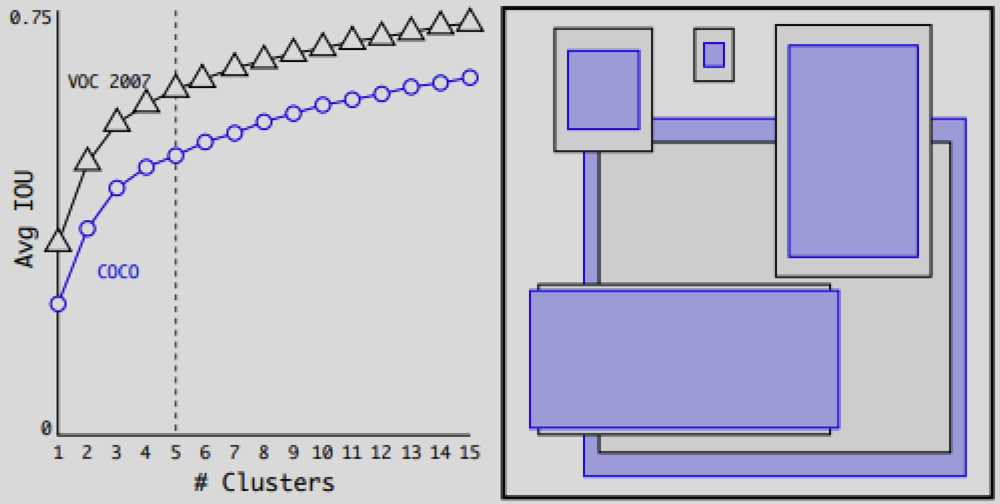
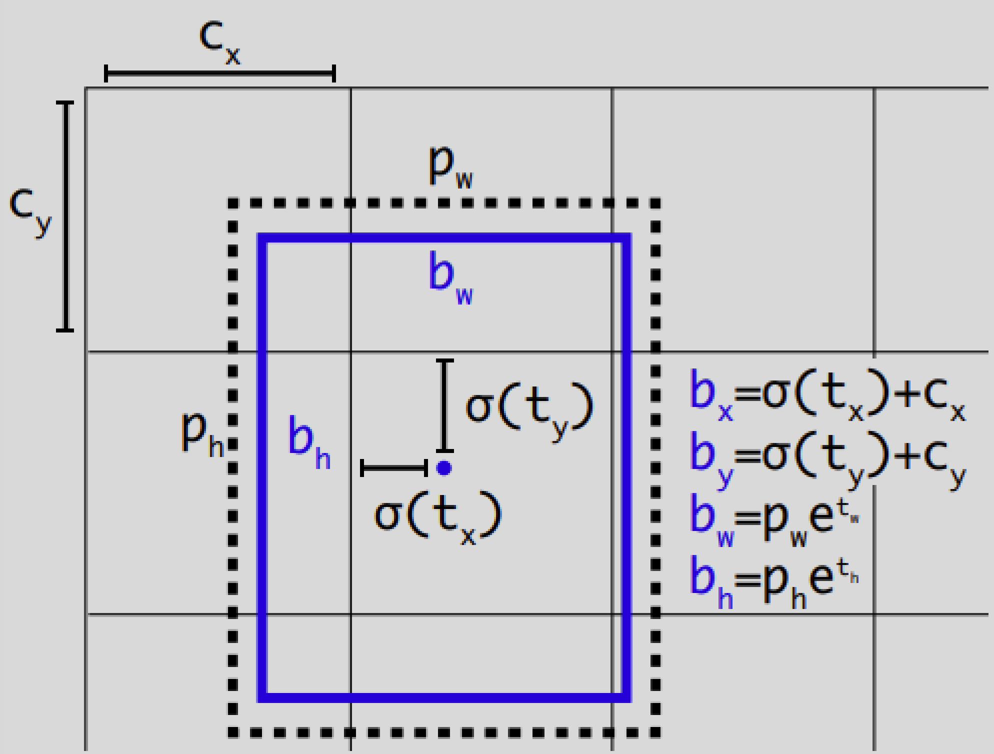


Figure 2: Clustering box dimensions on VOC and COCO. We run k-means clustering on the dimensions of bounding boxes to get good priors for our model. The left image shows the average IOU we get with various choices for k. We find that $k = 5$ gives a good tradeoff for recall vs. complexity of the model. The right image shows the relative centroids for VOC and COCO. Both sets of priors favor thinner, taller boxes while COCO has greater variation in size than VOC.

Location



$$b_x = \sigma(t_x) + c_x$$
$$b_y = \sigma(t_y) + c_y$$
$$b_w = p_w e^{t_w}$$
$$b_h = p_h e^{t_h}$$
$$Pr(\text{object}) * IOU(b, \text{object}) = \sigma(t_o)$$

Multi-scale training

- The original YOLO uses an input resolution 448x448.
- YOLOv2 uses the resolution to 416x416.
- Instead of fixing input image size, we change the network input every 10 batches. The possible input size is: {320, 352, …, 608}.

New network

- Similar to VGG network, use 3x3 filters and double the channels after pooling step.
- Use batch normalization to stabilizing training.

Type	Filters	Size/Stride	Output
Convolutional	32	3×3	224×224
Maxpool		$2 \times 2/2$	112×112
Convolutional	64	3×3	112×112
Maxpool		$2 \times 2/2$	56×56
Convolutional	128	3×3	56×56
Convolutional	64	1×1	56×56
Convolutional	128	3×3	56×56
Maxpool		$2 \times 2/2$	28×28
Convolutional	256	3×3	28×28
Convolutional	128	1×1	28×28
Convolutional	256	3×3	28×28
Maxpool		$2 \times 2/2$	14×14
Convolutional	512	3×3	14×14
Convolutional	256	1×1	14×14
Convolutional	512	3×3	14×14
Convolutional	256	1×1	14×14
Convolutional	512	3×3	14×14
Maxpool		$2 \times 2/2$	7×7
Convolutional	1024	3×3	7×7
Convolutional	512	1×1	7×7
Convolutional	1024	3×3	7×7
Convolutional	512	1×1	7×7
Convolutional	1024	3×3	7×7
Convolutional	1000	1×1	7×7
Avgpool		Global	1000
Softmax			

Table 6: Darknet-19.

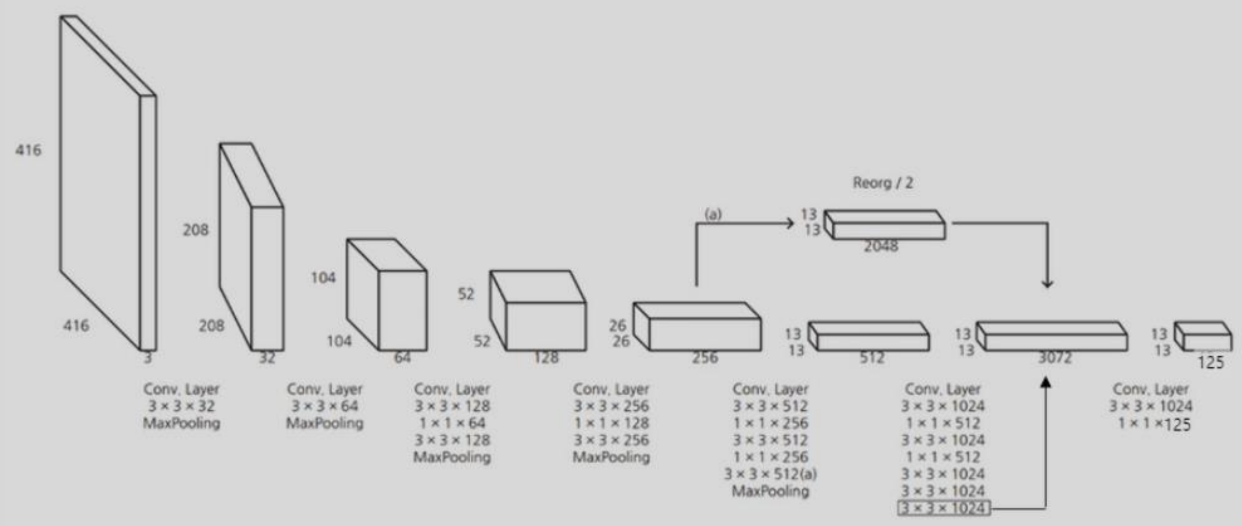
Multi-scale training

Detection Frameworks	Train	mAP	FPS
Fast R-CNN [5]	2007+2012	70.0	0.5
Faster R-CNN VGG-16[15]	2007+2012	73.2	7
Faster R-CNN ResNet[6]	2007+2012	76.4	5
YOLO [14]	2007+2012	63.4	45
SSD300 [11]	2007+2012	74.3	46
SSD500 [11]	2007+2012	76.8	19
YOLOv2 288×288	2007+2012	69.0	91
YOLOv2 352×352	2007+2012	73.7	81
YOLOv2 416×416	2007+2012	76.8	67
YOLOv2 480×480	2007+2012	77.8	59
YOLOv2 544×544	2007+2012	**78.6**	40

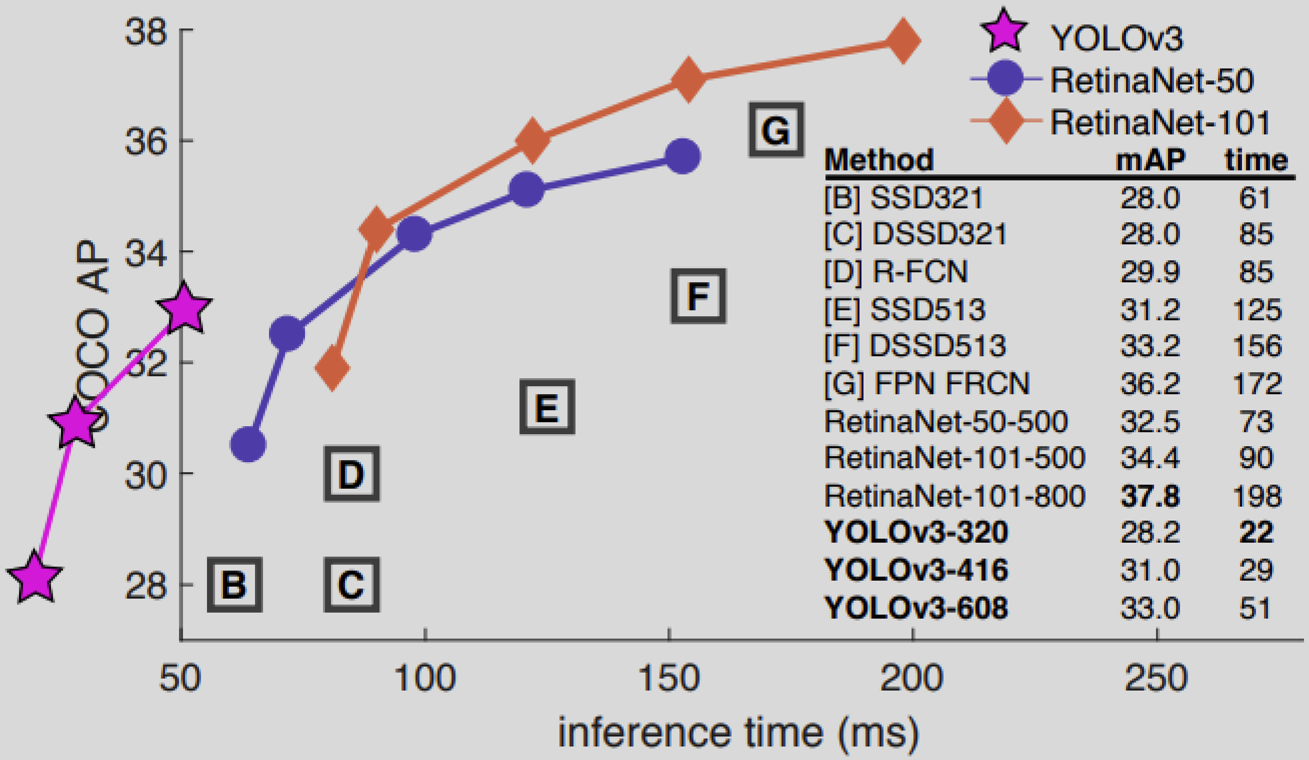
Passthrough layer

- Passthrough layer concatenates the higher resolution features with the low resolution features.

- This simple scheme improves the 1% performance increase.



YOLOv3



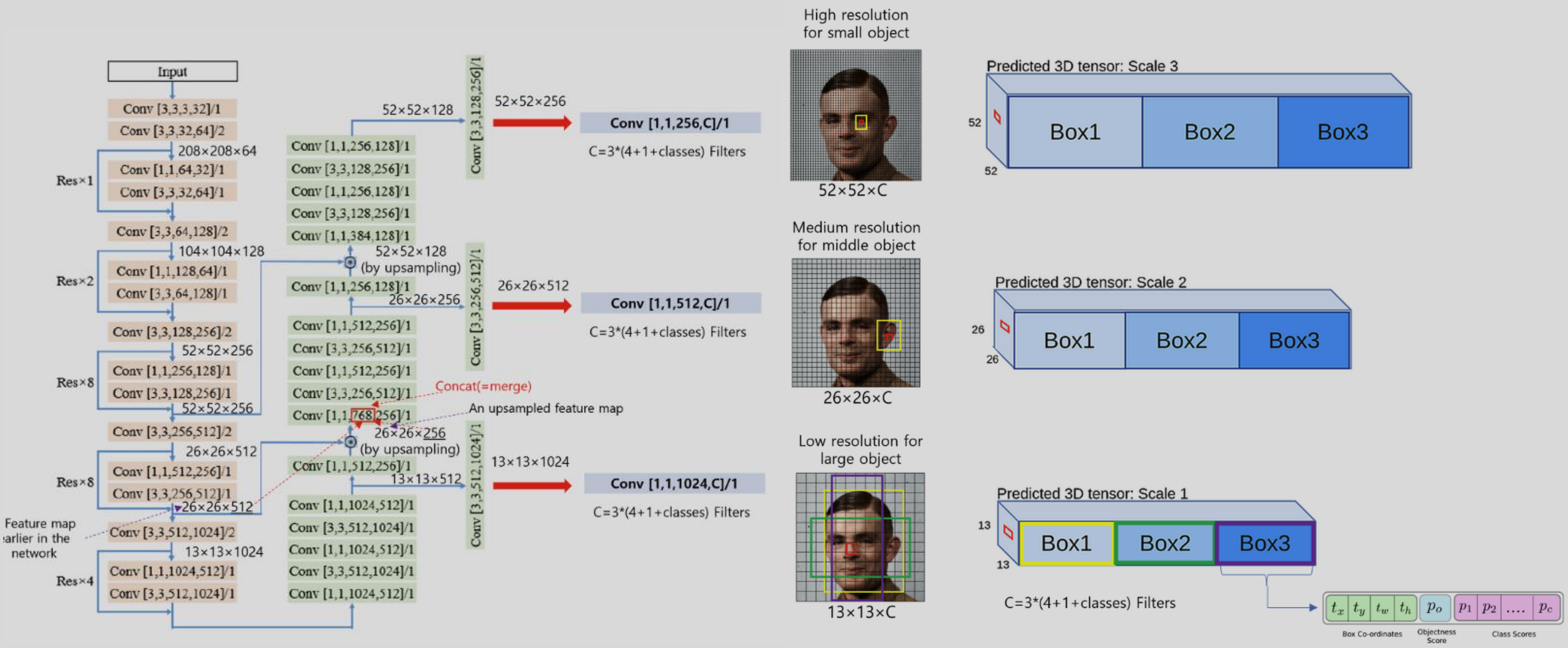
Method	mAP	time
[B] SSD321	28.0	61
[C] DSSD321	28.0	85
[D] R-FCN	29.9	85
[E] SSD513	31.2	125
[F] DSSD513	33.2	156
[G] FPN FRCN	36.2	172
RetinaNet-50-500	32.5	73
RetinaNet-101-500	34.4	90
RetinaNet-101-800	**37.8**	198
YOLOv3-320	28.2	**22**
YOLOv3-416	31.0	29
YOLOv3-608	33.0	51

Figure 1. We adapt this figure from the Focal Loss paper [9]. YOLOv3 runs significantly faster than other detection methods with comparable performance. Times from either an M40 or Titan X, they are basically the same GPU.

Multi-scale

- YOLOv3 predicts boxes at 3 different scales:
- NxNx[3*(4+1+80)]-dimensional array is predicted for 4 bounding box offsets, 1 objectness prediction and 80 class predictions in 3 different scales.

Multi-scale


Input
Conv [3,3,32]/1
Conv [3,3,32,64]/2
208×208×64
Res×1
Conv [1,1,64,32]/1
Conv [3,3,32,64]/1
Conv [3,3,64,128]/2
104×104×128
Res×2
Conv [1,1,128,64]/1
Conv [3,3,64,128]/1
Conv [3,3,128,256]/2
52×52×256
Res×8
Conv [1,1,256,128]/1
Conv [3,3,128,256]/1
52×52×256
Conv [3,3,256,512]/2
26×26×512
Res×8
Conv [1,1,512,256]/1
Conv [3,3,256,512]/1
26×26×512
Feature map earlier in the network
Conv [3,3,512,1024]/2
13×13×1024
Res×4
Conv [1,1,1024,512]/1
Conv [3,3,512,1024]/1
52×52×128
Conv [1,1,256,128]/1
Conv [3,3,128,256]/1
Conv [1,1,256,128]/1
Conv [3,3,128,256]/1
Conv [1,1,384,128]/1
52×52×128 (by upsampling)
Conv [1,1,256,128]/1
26×26×256
Conv [1,1,512,256]/1
Conv [3,3,256,512]/1
Conv [1,1,512,256]/1
Conv [3,3,256,512]/1
Conv [1,1,768,256]/1
Concat(=merge)
An upsampled feature map
26×26×256 (by upsampling)
Conv [1,1,512,256]/1
13×13×512
Conv [1,1,1024,512]/1
Conv [3,3,512,1024]/1
Conv [1,1,1024,512]/1
Conv [3,3,512,1024]/1
Conv [1,1,1024,512]/1
Conv [3,3,128,256]/1
Conv [3,3,256,512]/1
Conv [3,3,512,1024]/1
52×52×256
Conv [1,1,256,C]/1
C=3*(4+1+classes) Filters
26×26×512
Conv [1,1,512,C]/1
C=3*(4+1+classes) Filters
13×13×1024
Conv [1,1,1024,C]/1
C=3*(4+1+classes) Filters
High resolution for small object
52×52×C
Medium resolution for middle object
26×26×C
Low resolution for large object
13×13×C
Predicted 3D tensor: Scale 3
52
52
Box1
Box2
Box3
Predicted 3D tensor: Scale 2
26
26
Box1
Box2
Box3
Predicted 3D tensor: Scale 1
13
13
Box1
Box2
Box3
C=3*(4+1+classes) Filters
t_x t_y t_w t_h p_o p_1 p_2 .... p_c
Box Co-ordinates
Objectness Score
Class Scores

DarkNet-53

- Powerful than Darknet-19 but still more efficient than ResNet-101 or ResNet-152.

Backbone	Top-1	Top-5	Bn Ops	BFLOP/s	FPS
Darknet-19 [15]	74.1	91.8	7.29	1246	**171**
ResNet-101[5]	77.1	93.7	19.7	1039	53
ResNet-152 [5]	**77.6**	**93.8**	29.4	1090	37
Darknet-53	77.2	**93.8**	18.7	**1457**	78

	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
1×	Convolutional	32	1 × 1	
1×	Convolutional	64	3 × 3	
1×	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
2×	Convolutional	64	1 × 1	
2×	Convolutional	128	3 × 3	
2×	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
8×	Convolutional	128	1 × 1	
8×	Convolutional	256	3 × 3	
8×	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
8×	Convolutional	256	1 × 1	
8×	Convolutional	512	3 × 3	
8×	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
4×	Convolutional	512	1 × 1	
4×	Convolutional	1024	3 × 3	
4×	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

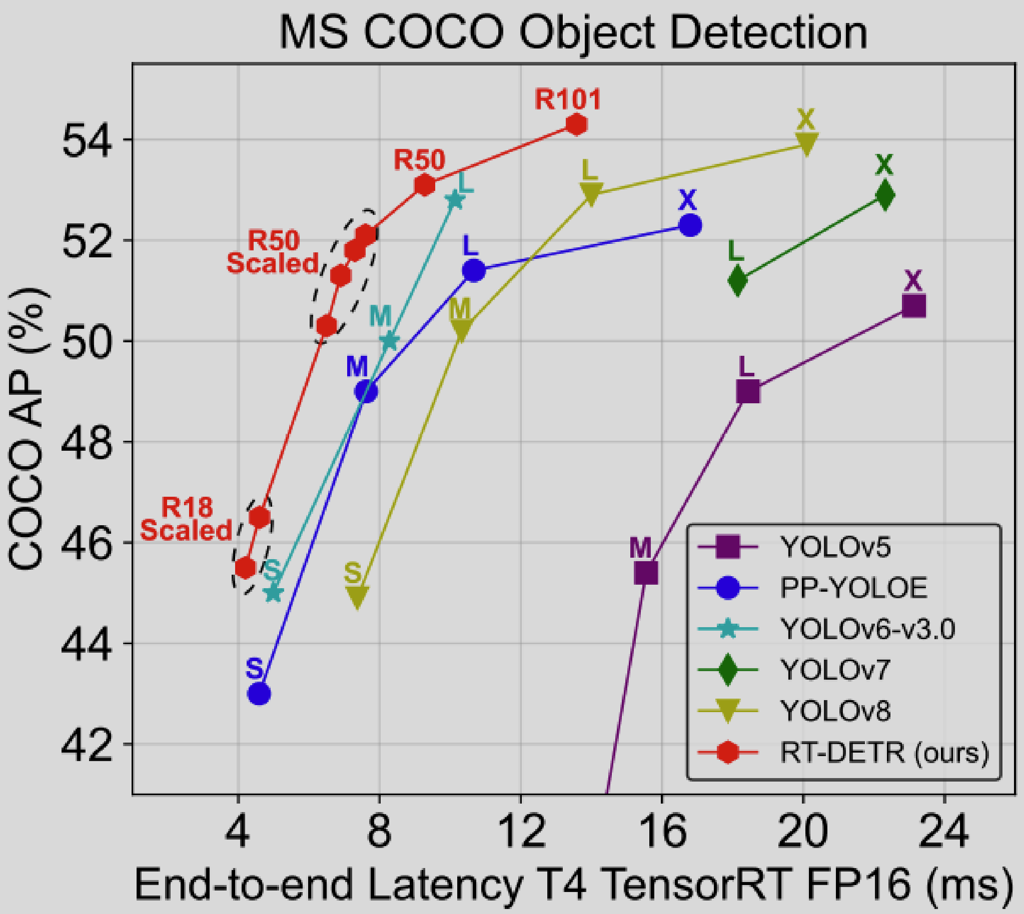
Table 1. **Darknet-53.**

RT-DETR

DETRs Beat YOLOs on Real-time Object Detection

Wenyu Lv, Yian Zhao, Shangliang Xu, Jinman Wei, Guanzhong Wang, Cheng Cui
Yuning Du, Qingqing Dang, Yi Liu
Baidu Inc.
{lvwenyu01, zhaoyian, xushangliang, wangguanzhong} @baidu.com

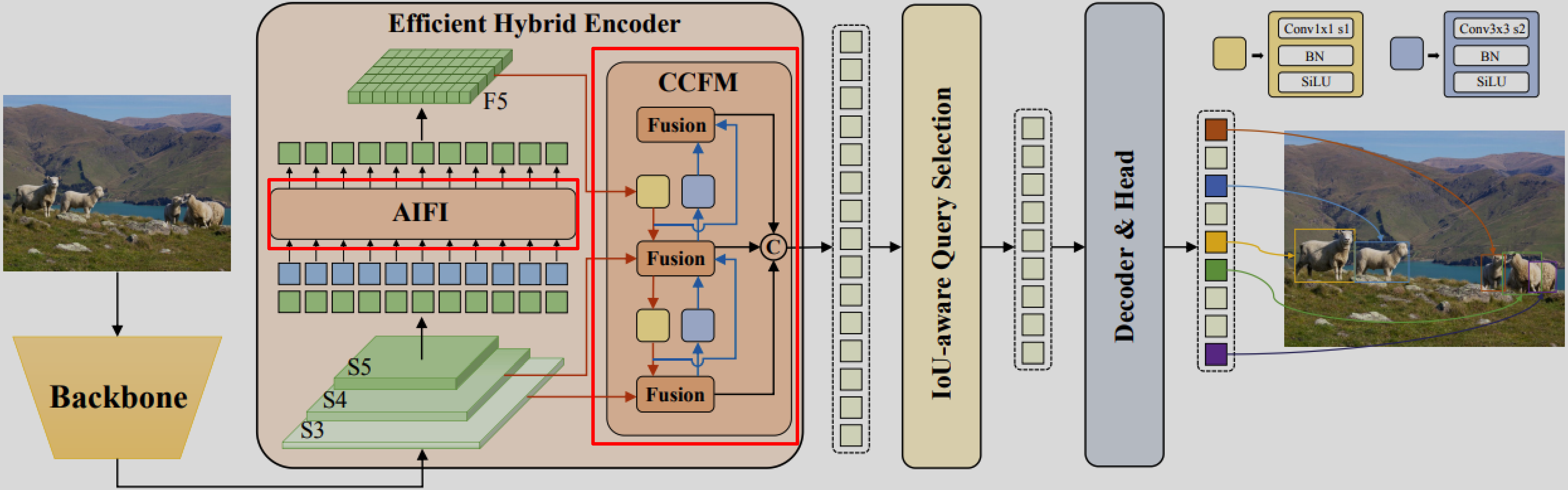
RT-DETR


MS COCO Object Detection
COCO AP (%)
End-to-end Latency T4 TensorRT FP16 (ms)
R101
R50
R50 Scaled
R18 Scaled
YOLOv5
PP-YOLOE
YOLOv6-v3.0
YOLOv7
YOLOv8
RT-DETR (ours)

RT-DETR

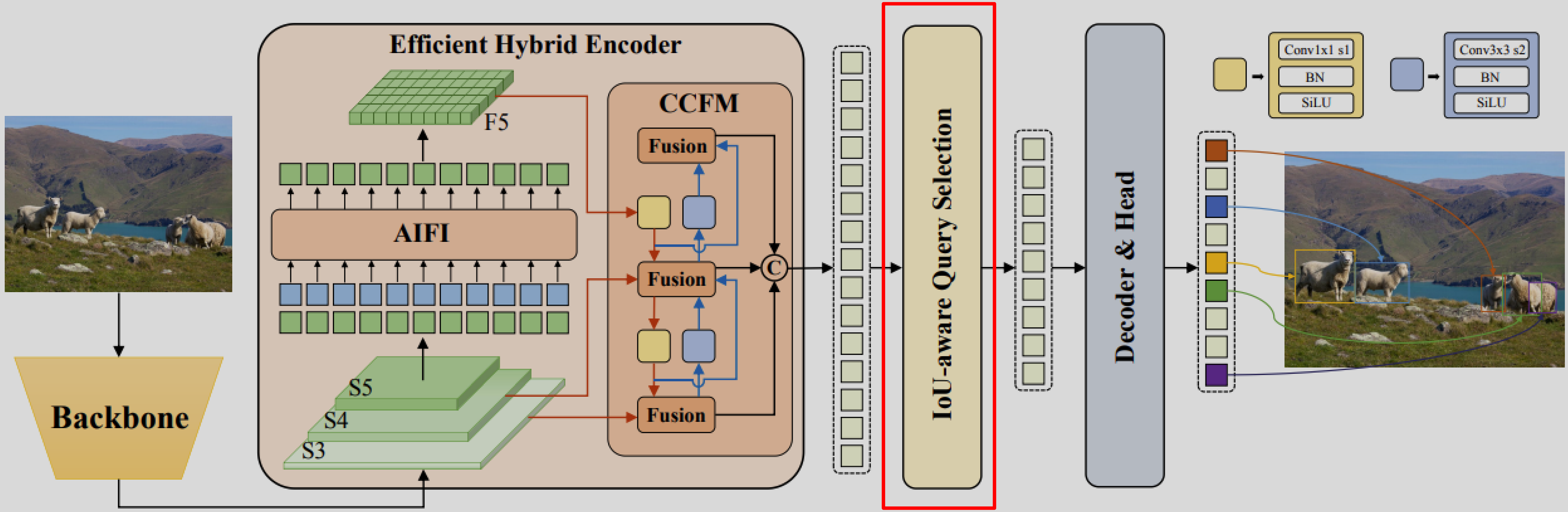
- Multi-scale features are helpful for improving the performance.
- However, it hinders the real-time speed.
- Encoder accounts for 49% of the overall GFLOPs,
  but contributes only 11% of the AP in Deformable DETR.

RT-DETR



- Propose the efficient hybrid encoder: having attention-based intra-scale feature interaction (AIFI), cross-scale feature-fusion module (CCFM).

RT-DETR



- IoU-aware query selection: produce high classification scores for features with high IoU scores.
- No need to enforce high classification scores for features with low IoU scores.

RT-DETR

Model	Backbone	#Epochs	#Params (M)	GFLOPs	$FPS_{bs=1}$	AP^{val}	AP^{val}_{50}	AP^{val}_{75}	AP^{val}_{S}	AP^{val}_{M}	AP^{val}_{L}
Real-time Object Detectors											
YOLOv5-L[13]	-	300	46	109	54	49.0	67.3	-	-	-	-
YOLOv5-X[13]	-	300	86	205	43	50.7	68.9	-	-	-	-
PPYOLOE-L[41]	CSPRepResNet	300	52	110	94	51.4	68.9	55.6	31.4	55.3	66.1
PPYOLOE-X[41]	CSPRepResNet	300	98	206	60	52.3	69.9	56.5	33.3	56.3	66.4
YOLOv6-L[18]	-	300	59	150	99	52.8	70.3	57.7	34.4	58.1	70.1
YOLOv7-L[38]	-	300	36	104	55	51.2	69.7	55.5	35.2	55.9	66.7
YOLOv7-X[38]	-	300	71	189	45	52.9	71.1	57.4	36.9	57.7	68.6
YOLOv8-L[14]	-	-	43	165	71	52.9	69.8	57.5	35.3	58.3	69.8
YOLOv8-X[14]	-	-	68	257	50	53.9	71.0	58.7	35.7	59.3	70.7
End-to-end Object Detectors											
DETR-DC5 [5]	R50	500	41	187	-	43.3	63.1	45.9	22.5	47.3	61.1
DETR-DC5 [5]	R101	500	60	253	-	44.9	64.7	47.7	23.7	49.5	62.3
Anchor-DETR-DC5 [40]	R50	50	39	172	-	44.2	64.7	47.5	24.7	48.2	60.6
Anchor-DETR-DC5 [40]	R101	50	-	-	-	45.1	65.7	48.8	25.8	49.4	61.6
Conditional-DETR-DC5 [27]	R50	108	44	195	-	45.1	65.4	48.5	25.3	49.0	62.2
Conditional-DETR-DC5 [27]	R101	108	63	262	-	45.9	66.8	49.5	27.2	50.3	63.3
Efficient-DETR [43]	R50	36	35	210	-	45.1	63.1	49.1	28.3	48.4	59.0
Efficient-DETR [43]	R101	36	54	289	-	45.7	64.1	49.5	28.2	49.1	60.2
SMCA-DETR [9]	R50	108	40	152	-	45.6	65.5	49.1	25.9	49.3	62.6
SMCA-DETR [9]	R101	108	58	218	-	46.3	66.6	50.2	27.2	50.5	63.2
Deformable-DETR [49]	R50	50	40	173	-	46.2	65.2	50.0	28.8	49.2	61.7
DAB-Deformable-DETR [24]	R50	50	48	195	-	46.9	66.0	50.8	30.1	50.4	62.5
DN-Deformable-DETR [20]	R50	50	48	195	-	48.6	67.4	52.7	31.0	52.0	63.7
DAB-Deformable-DETR++ [20]	R50	50	47	-	-	48.7	67.2	53.0	31.4	51.6	63.9
DN-Deformable-DETR++ [20]	R50	50	47	-	-	49.5	67.6	53.8	31.3	52.6	65.4
DINO-Deformable-DETR [46]	R50	36	47	279	5	50.9	69.0	55.3	34.6	54.1	64.6
Real-time End-to-end Object Detector (ours)											
RT-DETR-R50	R50	72	42	136	**108**	**53.1**	**71.3**	**57.7**	34.8	58.0	70.0
RT-DETR-R101	R101	72	76	259	**74**	**54.3**	**72.7**	58.6	36.0	58.8	**72.1**
RT-DETR-L	HGNetv2	72	32	110	**114**	**53.0**	**71.6**	57.3	34.6	57.3	**71.2**
RT-DETR-X	HGNetv2	72	67	234	**74**	**54.8**	**73.1**	**59.4**	35.7	**59.6**	**72.9**

Thank you!

UNIST

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

2 0 0 7