```python
# Define the window size
window_size = (64, 64)

# Define the step size
step_size = (8, 8)

# Define how many steps to take in x and y directions
x_steps = np.arange(0, image.shape[1] - window_size[1], step_size[0])
y_steps = np.arange(0, image.shape[0] - window_size[0], step_size[1])
```

8

image.shape[1] - window_size[1]

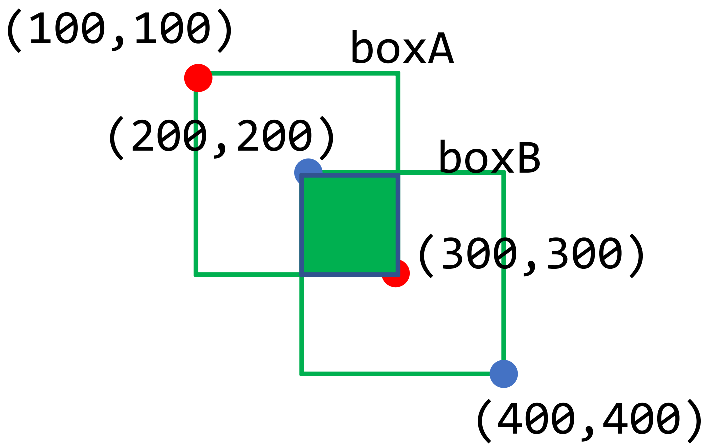
64

64

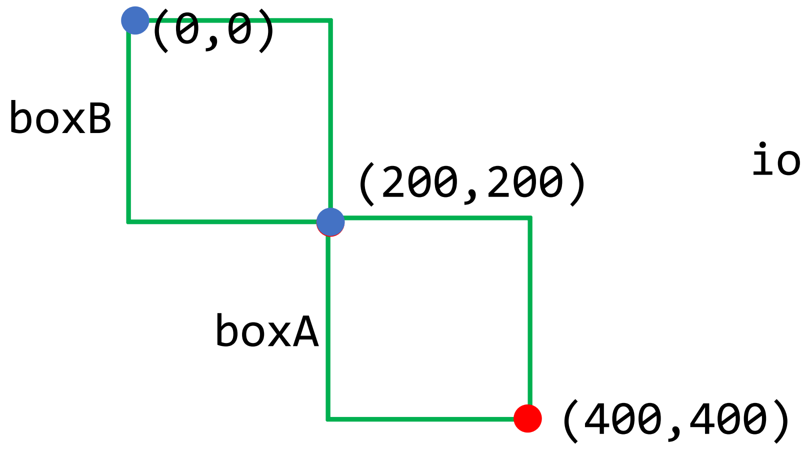
height
== image.shape[0]

width==image.shape[1]

```
def compute_iou(boxA, boxB):
    xA = max(boxA[0], boxB[0])  # xA = 200
    yA = max(boxA[1], boxB[1])  # yA = 200
    xB = min(boxA[2], boxB[2])  # xB = 300
    yB = min(boxA[3], boxB[3])  # yB = 300

    interArea = max(0, xB - xA + 1) * max(0, yB - yA + 1)

    boxAArea = (boxA[2] - boxA[0] + 1) * (boxA[3] - boxA[1] + 1)
    boxBArea = (boxB[2] - boxB[0] + 1) * (boxB[3] - boxB[1] + 1)

    iou = interArea / float(boxAArea + boxBArea - interArea)

    return iou
```

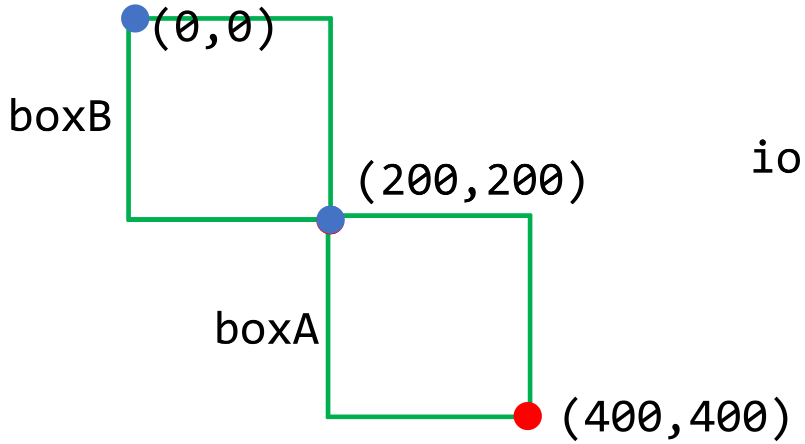


```python
def compute_iou(boxA, boxB):
    xA = max(boxA[0], boxB[0])  # xA = 200
    yA = max(boxA[1], boxB[1])  # yA = 200
    xB = min(boxA[2], boxB[2])  # xB = 200
    yB = min(boxA[3], boxB[3])  # yB = 200

    interArea = max(0, xB - xA + 1) * max(0, yB - yA + 1)

    boxAArea = (boxA[2] - boxA[0] + 1) * (boxA[3] - boxA[1] + 1)
    boxBArea = (boxB[2] - boxB[0] + 1) * (boxB[3] - boxB[1] + 1)

    iou = interArea / float(boxAArea + boxBArea - interArea)

    return iou
```



iou = 0.0000123...

```python
def compute_iou(boxA, boxB):
    xA = max(boxA[0], boxB[0])  # xA = 200
    yA = max(boxA[1], boxB[1])  # yA = 200
    xB = min(boxA[2], boxB[2])  # xB = 200
    yB = min(boxA[3], boxB[3])  # yB = 200

    interArea = max(0, xB - xA + 1) * max(0, yB - yA + 1)

    boxAArea = (boxA[2] - boxA[0] + 1) * (boxA[3] - boxA[1] + 1)
    boxBArea = (boxB[2] - boxB[0] + 1) * (boxB[3] - boxB[1] + 1)

    iou = interArea / float(boxAArea + boxBArea - interArea)

    return iou
```



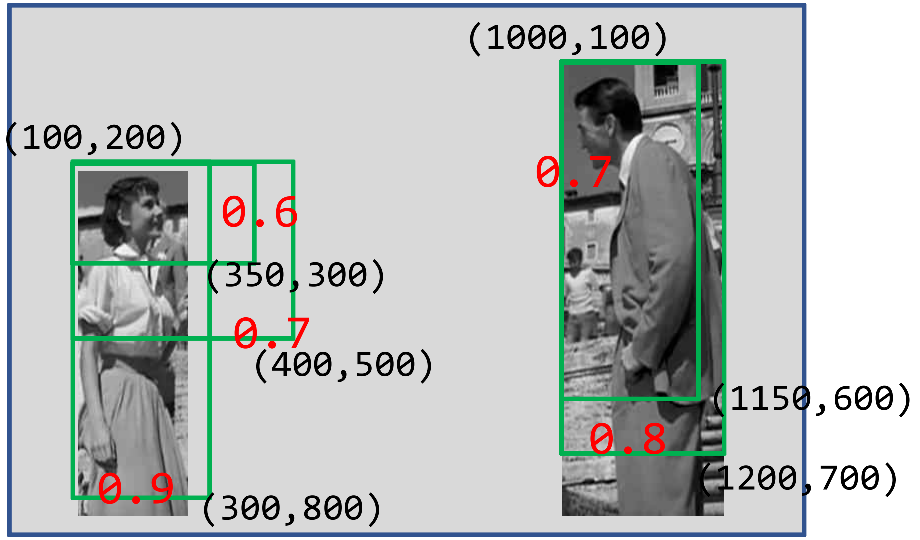
iou = 0.0000123...

```python
def non_max_suppression(boxes, probs=None, overlapThresh=0.3):
    if len(boxes) == 0:
        return []

    if boxes.dtype.kind == "i":
        boxes = boxes.astype("float")

    pick = []

    x1 = boxes[:, 0]  # [100,100,100,1000,1000]
    y1 = boxes[:, 1]  # [200,200,200,100,100]
    x2 = boxes[:, 2]  # [350,400,300,1150,1200]
    y2 = boxes[:, 3]  # [300,500,800,600,700]

    area = (x2 - x1 + 1) * (y2 - y1 + 1)
    idxs = probs

    idxs = np.argsort(idxs)
```

```
[ 100.   100.   100. 1000. 1000.]
[200. 200. 200. 100. 100.]
[ 350.   400.   300. 1150. 1200.]
[300. 500. 800. 600. 700.]
```

```python
boxes=np.array([
    [100,200,350,300],
    [100,200,400,500],
    [100,200,300,800],
    [1000,100,1150,600],
    [1000,100,1200,700]
])
probs = np.array([0.6,0.7,0.9,0.7,0.8])
```

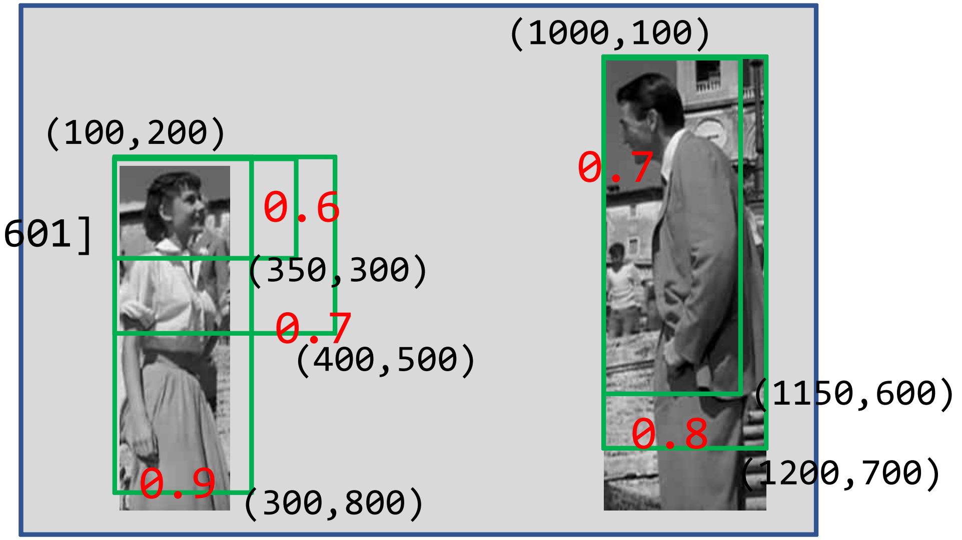
```
0.6 0.7 0.7 0.8 0.9
 0   1   3   4   2
```



(251,101)

```
  xx1 = np.maximum(x1[i], x1[idxs[:last]])
#                [100,100,100,100]   [100,100,1000,1000]
# print(xx1)  # [100,100,1000,1000]
  yy1 = np.maximum(y1[i], y1[idxs[:last]])
# print(yy1)  # [200,200,200,200]
  xx2 = np.minimum(x2[i], x2[idxs[:last]])
# print(xx2)  # [300,300,300,300]
  yy2 = np.minimum(y2[i], y2[idxs[:last]])
# print(yy2)  # [300,500,600,700]

  w = np.maximum(0, xx2 - xx1 + 1)
print(w)  #  [0,0,0,0]   [201,201,-699,-699] => [201,201,0,0]
  h = np.maximum(0, yy2 - yy1 + 1)
print(w)  #  [0,0,0,0]   [101,301,401,501] => [101,301,401,501]
print(area[idxs[:last]]) # [251*101,301*301,151*501,201*601]
  overlap = (w * h) / area[idxs[:last]]

print(overlap)  # [20301,60501,0,0] / [251*101,301*301,151*501,201*601]
                # [0.80,0.67,0,0]

print(np.where(overlap > overlapThresh)[0])  # [0,1]
print(np.concatenate(([last],np.where(overlap > overlapThresh)[0]))
                                             # [4,0,1]
  idxs = np.delete(idxs, np.concatenate(([last],
                 np.where(overlap > overlapThresh)[0])))

eturn pick
```

```
boxes=np.array([
   [100,200,350,300],
   [100,200,400,500],
   [100,200,300,800],
   [1000,100,1150,600],
   [1000,100,1200,700]
])
probs = np.array([0.6,0.7,0.9,0.7,0.8])
                  0.6 0.7 0.7 0.8 0.9
                   0   1   3   4   2
```



	last
idxs	0 · 1 · 3 · 4 · 2
pick	2

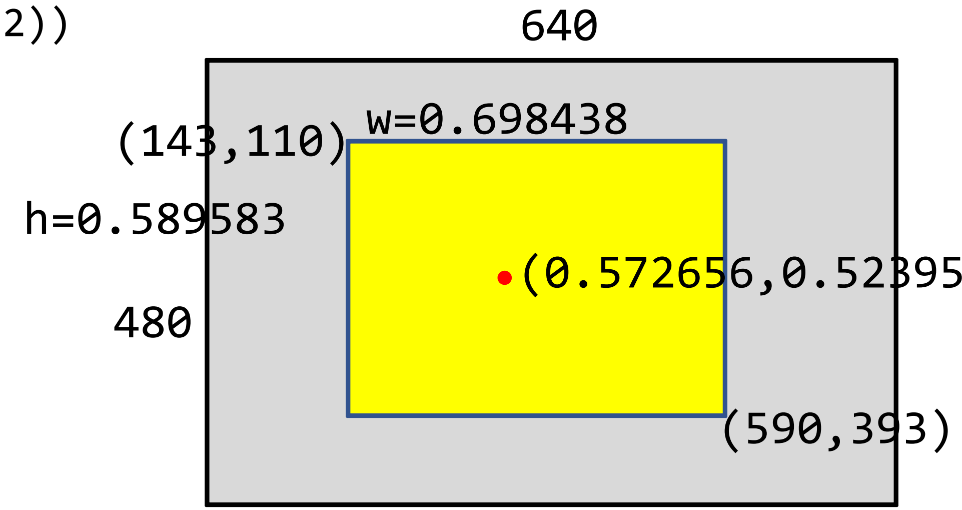
	last
idxs	3 · 4
pick	2 · 4

```
# box : (centerX, centerY, width, height)
def convertToAbsoluteValues(size, box):

    xIn = round(((2 * float(box[0]) - float(box[2])) * size[0] / 2))
    yIn = round(((2 * float(box[1]) - float(box[3])) * size[1] / 2))
    xEnd = xIn + round(float(box[2]) * size[0])
    yEnd = yIn + round(float(box[3]) * size[1])

    if xIn < 0:
        xIn = 0
    if yIn < 0:
        yIn = 0
    if xEnd >= size[0]:
        xEnd = size[0] - 1
    if yEnd >= size[1]:
        yEnd = size[1] - 1
    return (xIn, yIn, xEnd, yEnd)
```

DT=(0.657031, 0.653125, 0.648438,

GT=(0.572656, 0.523958, 0.698438,

size=(640,480)



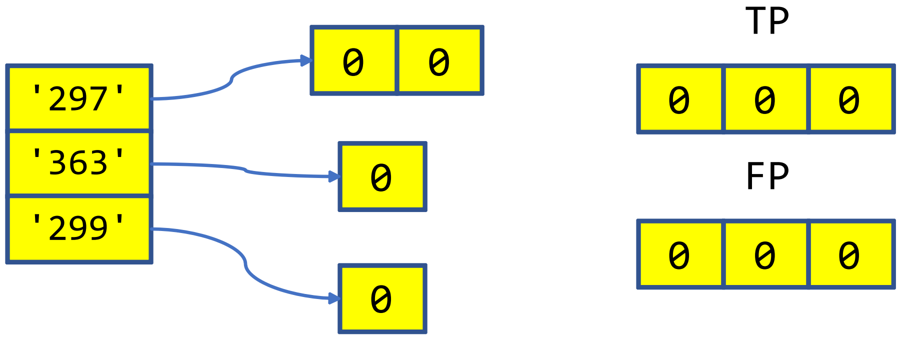
```
def AP(detections, groundtruths, classes, IOUThreshold = 0.3, method = 'AP'):

    result = []

    for c in classes:
        dects = [d for d in detections if d[1] == c]
        gts = [g for g in groundtruths if g[1] == c]
        dects = sorted(dects, key = lambda conf : conf[2], reverse=True)
        TP = np.zeros(len(dects))
        FP = np.zeros(len(dects))
        det = Counter(cc[0] for cc in gts)
        for key, val in det.items():
            det[key] = np.zeros(val)
```

dects

```
[['363', 0.0, 0.92, (280, 69, 696, 435)],
 ['297', 0.0, 0.77, (499, 286, 791, 455)],
 ['297', 0.0, 0.33, (138, 173, 335, 330)]]
```

dects

```
[['297', 0.0, 0.33, (138, 173, 335, 330)],
 ['297', 0.0, 0.77, (499, 286, 791, 455)],
 ['363', 0.0, 0.92, (280, 69, 696, 435)]]
```



```python
for d in range(len(dects)):
    gt = [gt for gt in gts if gt[0] == dects[d][0]]
    iouMax = 0

    for j in range(len(gt)):
        iou1 = iou(dects[d][3], gt[j][3])
        if iou1 > iouMax:
            iouMax = iou1
            jmax = j

    if iouMax >= IOUThreshold:
        if det[dects[d][0]][jmax] == 0:
            TP[d] = 1
            det[dects[d][0]][jmax] = 1
        else:
            FP[d] = 1
    else:
        FP[d] = 1
```

dects

```
[['363', 0.0, 0.92, (280, 69, 696, 435)],
 ['297', 0.0, 0.77, (499, 286, 791, 455)],
 ['297', 0.0, 0.33, (138, 173, 335, 330)]]
```

gts

```
[['297', 0.0, 1.0, (106, 195, 303, 352)],
 ['297', 0.0, 1.0, (548, 250, 841, 419)],
 ['363', 0.0, 1.0, (207, 108, 624, 474)],
 ['299', 0.0, 1.0, (434, 223, 715, 386)]]
```

gt

```
[['363', 0.0, 1.0, (207, 108, 624, 474)]]
```

det

'297'	0	0
'363'	1	
'299'	0	

d

TP

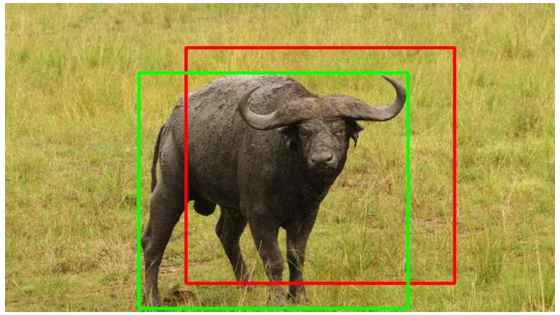
1	0	0

FP

0	0	0

iouMax	jmax
0.59	0

363



```
for d in range(len(dects)):
    gt = [gt for gt in gts if gt[0] == dects[d][0]]
    iouMax = 0

    for j in range(len(gt)):
        iou1 = iou(dects[d][3], gt[j][3])
        if iou1 > iouMax:
            iouMax = iou1
            jmax = j

    if iouMax >= IOUThreshold:
        if det[dects[d][0]][jmax] == 0:
            TP[d] = 1
            det[dects[d][0]][jmax] = 1
        else:
            FP[d] = 1
    else:
        FP[d] = 1
```

dects

```
[['363', 0.0, 0.92, (280, 69, 696, 435)],
 ['297', 0.0, 0.77, (499, 286, 791, 455)],
 ['297', 0.0, 0.33, (138, 173, 335, 330)]]
```

gts

```
[['297', 0.0, 1.0, (106, 195, 303, 352)],
 ['297', 0.0, 1.0, (548, 250, 841, 419)],
 ['363', 0.0, 1.0, (207, 108, 624, 474)],
 ['299', 0.0, 1.0, (434, 223, 715, 386)]]
```

gt

```
[['297', 0.0, 1.0, (106, 195, 303, 352)],
 ['297', 0.0, 1.0, (548, 250, 841, 419)]]
```

det

'297' → 1 1

'363' → 1

'299' → 0

d

TP

1 1 1

FP

0 0 0

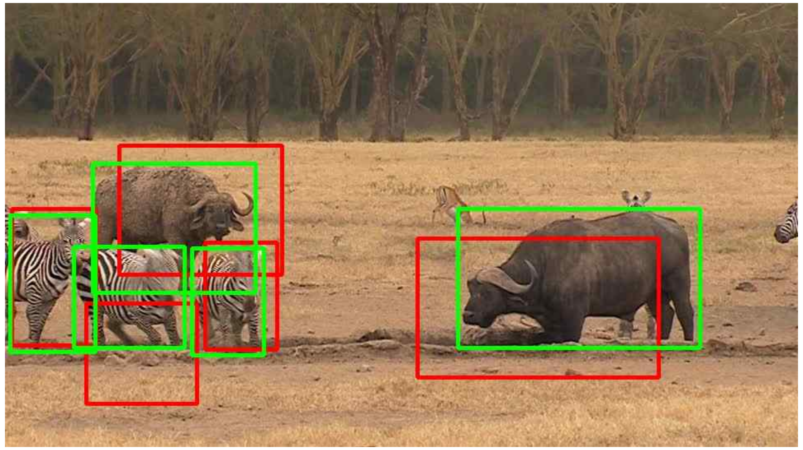
iouMax

0.56

jmax

0

297



```
acc_FP = np.cumsum(FP)
acc_TP = np.cumsum(TP)
rec = acc_TP / npos
prec = np.divide(acc_TP, (acc_FP + acc_TP))
```

dects

```
[['363', 0.0, 0.92, (280, 69, 696, 435)],
 ['297', 0.0, 0.77, (499, 286, 791, 455)],
 ['297', 0.0, 0.33, (138, 173, 335, 330)]]
```

acc_FP

0	0	0

acc_TP

1	2	3

rec

0.25	0.50	0.75

prec

1.0	1.0	1.0

gts

```
[['297', 0.0, 1.0, (106, 195, 303, 352)],
 ['297', 0.0, 1.0, (548, 250, 841, 419)],
 ['363', 0.0, 1.0, (207, 108, 624, 474)],
 ['299', 0.0, 1.0, (434, 223, 715, 386)]]
```

d

det

det	
'297'	1 1
'363'	1
'299'	0

TP

1	1	1

FP

0	0	0

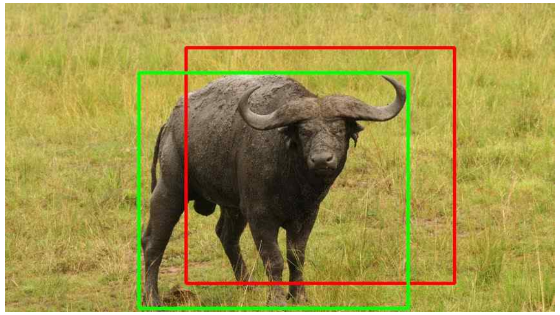
npos

4

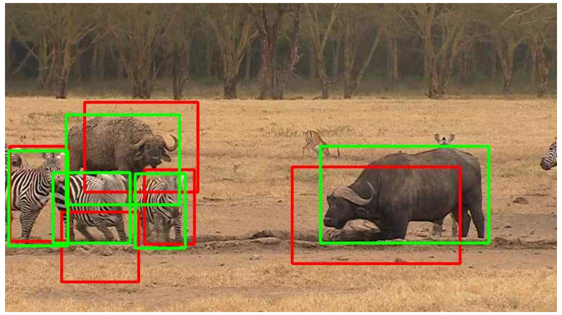
gt

```
[['297', 0.0, 1.0, (106, 195, 303, 352)],
 ['297', 0.0, 1.0, (548, 250, 841, 419)]]
```

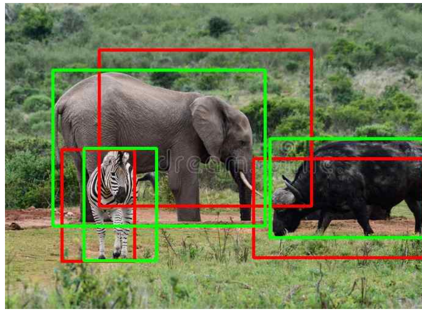
363



297



299



```
def calculateAveragePrecision(rec, prec):

    mrec = [0] + [e for e in rec] + [1]
    mpre = [0] + [e for e in prec] + [0]

    for i in range(len(mpre)-1, 0, -1):
        mpre[i-1] = max(mpre[i-1], mpre[i])

    ii = []

    for i in range(len(mrec)-1):
        if mrec[1:][i] != mrec[0:-1][i]:
            ii.append(i+1)

    ap = 0
    for i in ii:
        ap = ap + np.sum((mrec[i] - mrec[i-1]) * mpre[i])

    return [ap, mpre[0:len(mpre)-1], mrec[0:len(mpre)-1], ii]
```

rec

0.25	0.50	0.75

prec

1.0	1.0	1.0

mrec

0.0	0.25	0.50	0.75	1.0

mpre

1.0	1.0	1.0	1.0	0.0

0.25	0.50	0.75	0.75

0.0	0.25	0.50	0.75

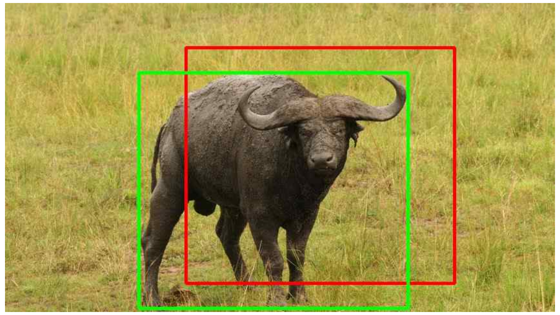
ii

1	2	3	4

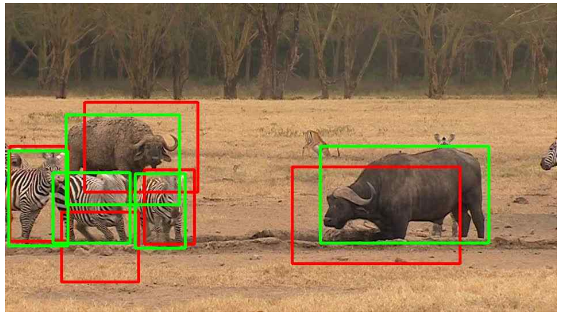
ap

0.75

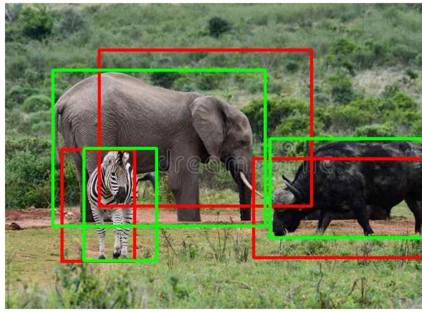
363

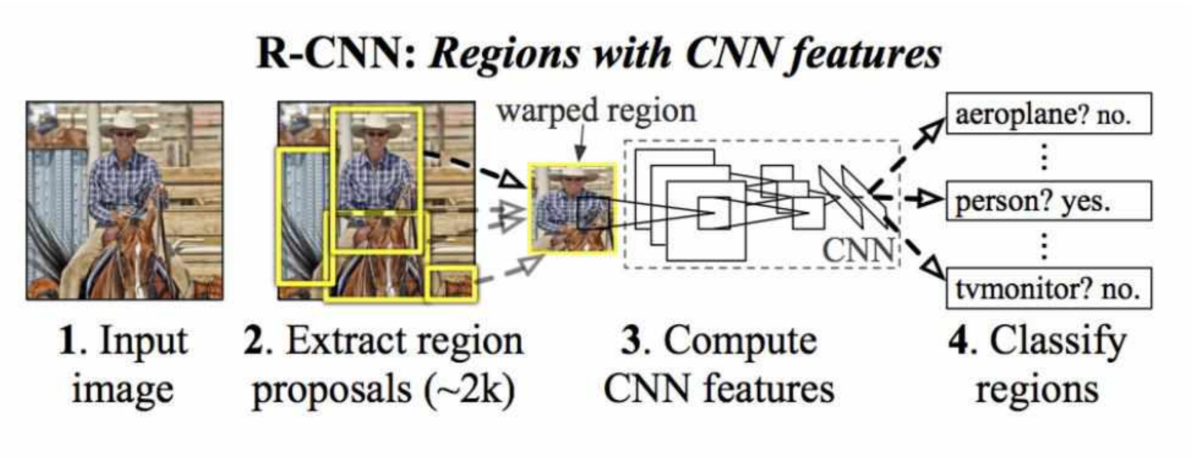


297

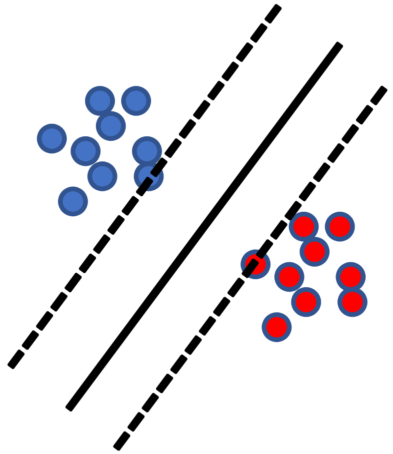


299





R-CNN

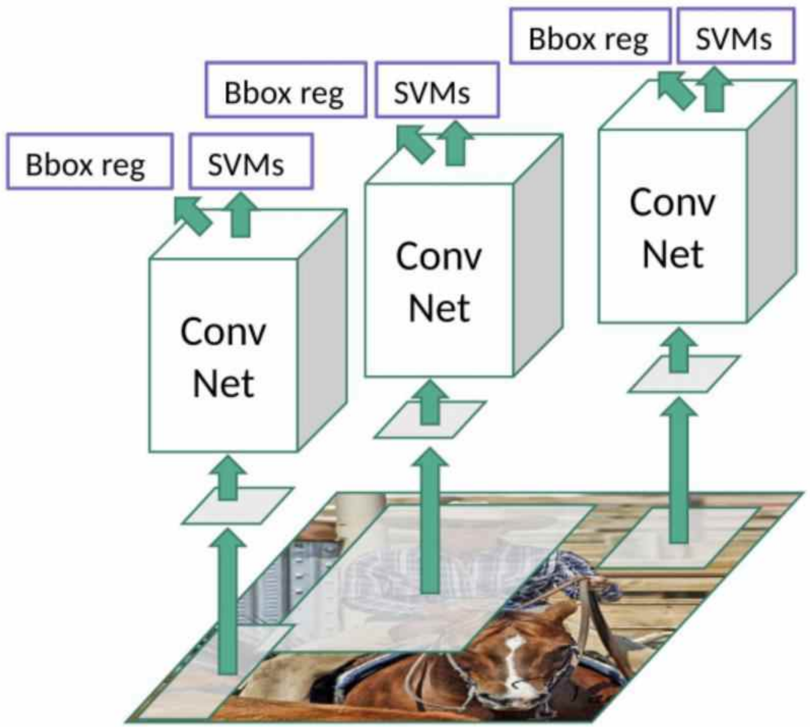


1. Selective search 알고리즘을 통해 객체가 있을 법할 위치인 후보 영역(region proposal)을 2000개 추출하여, 각각을 227x227 크기로 warp시켜줍니다.

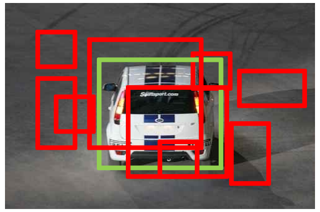
2. Warp된 모든 region proposal을 Fine tune된 AlexNet에 입력하여 2000x4096 크기의 feature vector를 추출합니다.

3. 추출된 feature vector를 linear SVM 모델과 Bounding box regressor 모델에 입력하여 각각 confidence score와 조정된 bounding box 좌표를 얻습니다.

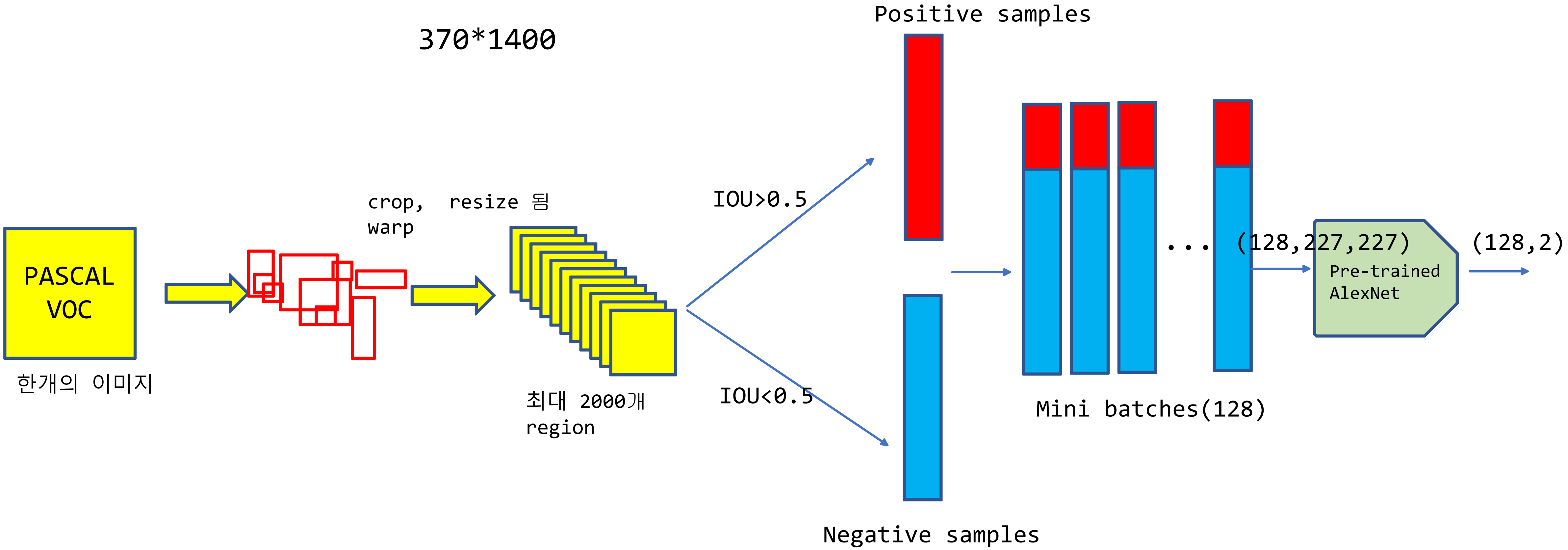
4. 마지막으로 Non maximum suppression 알고리즘을 적용하여 최소한의, 최적의 bounding box를 출력합니다.


Bbox reg
SVMs
Bbox reg
SVMs
Bbox reg
SVMs
Conv Net
Conv Net
Conv Net

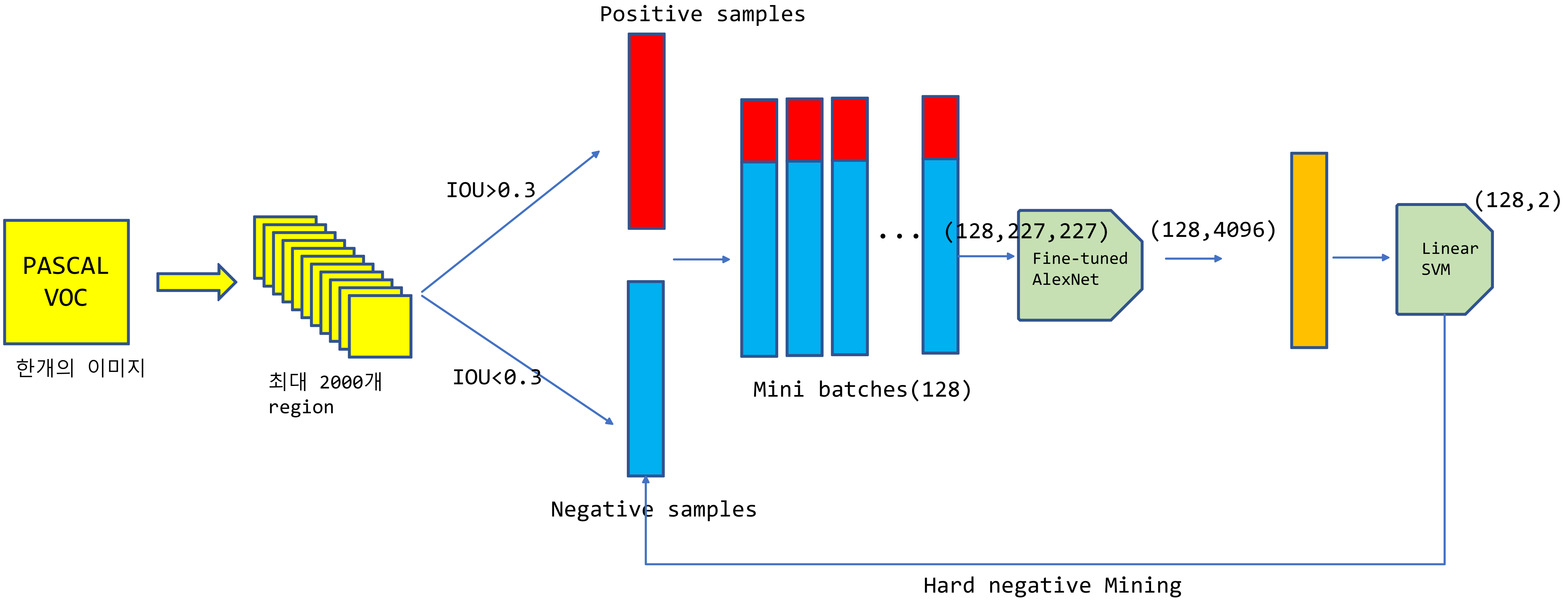
R-CNN



1. Fine tuning pre-trained AlexNet



2. Training linear SVM using fine tuned AlexNet



수정 예측 값

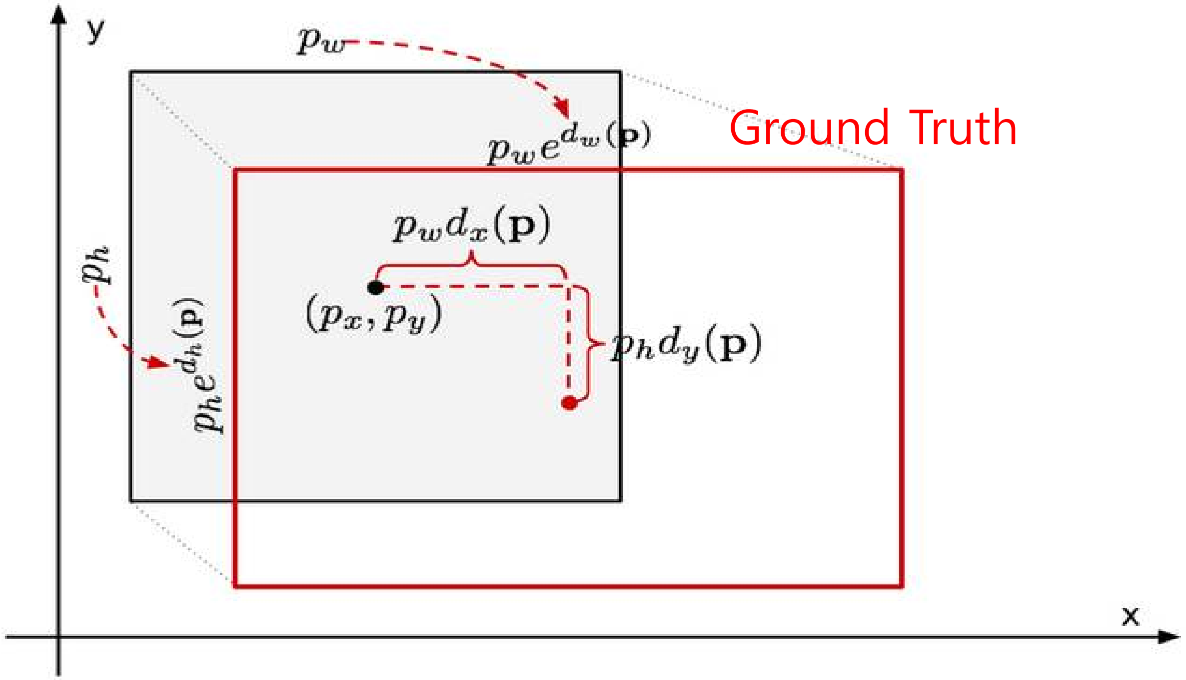
$$\hat{g}_x = p_w d_x(p) + p_x$$

$$\hat{g}_y = p_h d_y(p) + p_x$$

$$\hat{g}_w = p_w \exp(d_w(p))$$

$$\hat{g}_h = p_h \exp(d_h(p))$$

SS Predicted



Target

$$t_x = (g_x - p_x)/p_w$$

$$t_y = (g_y - p_y)/p_h$$

$$t_w = \log(g_w/p_w)$$

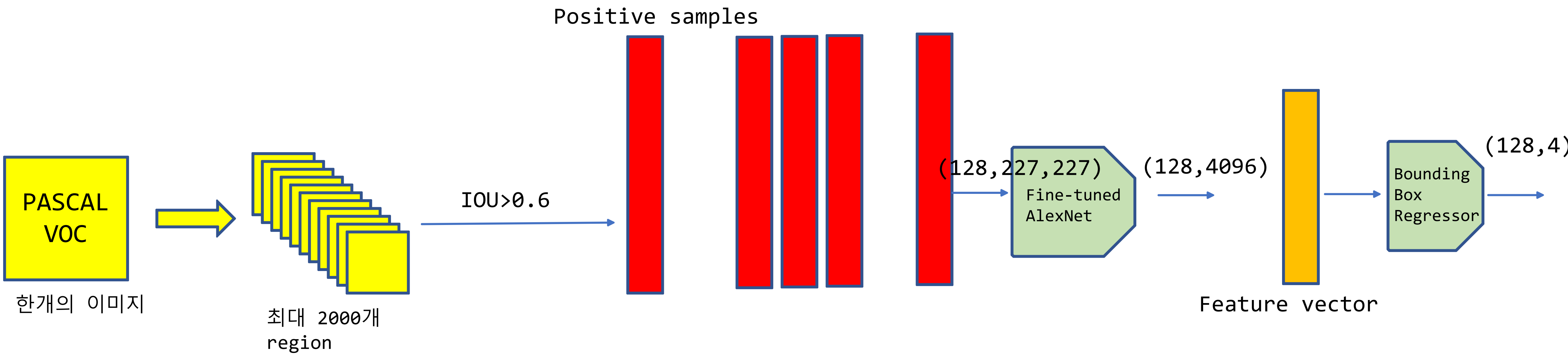
$$t_h = \log(g_h/h)$$

손실 함수

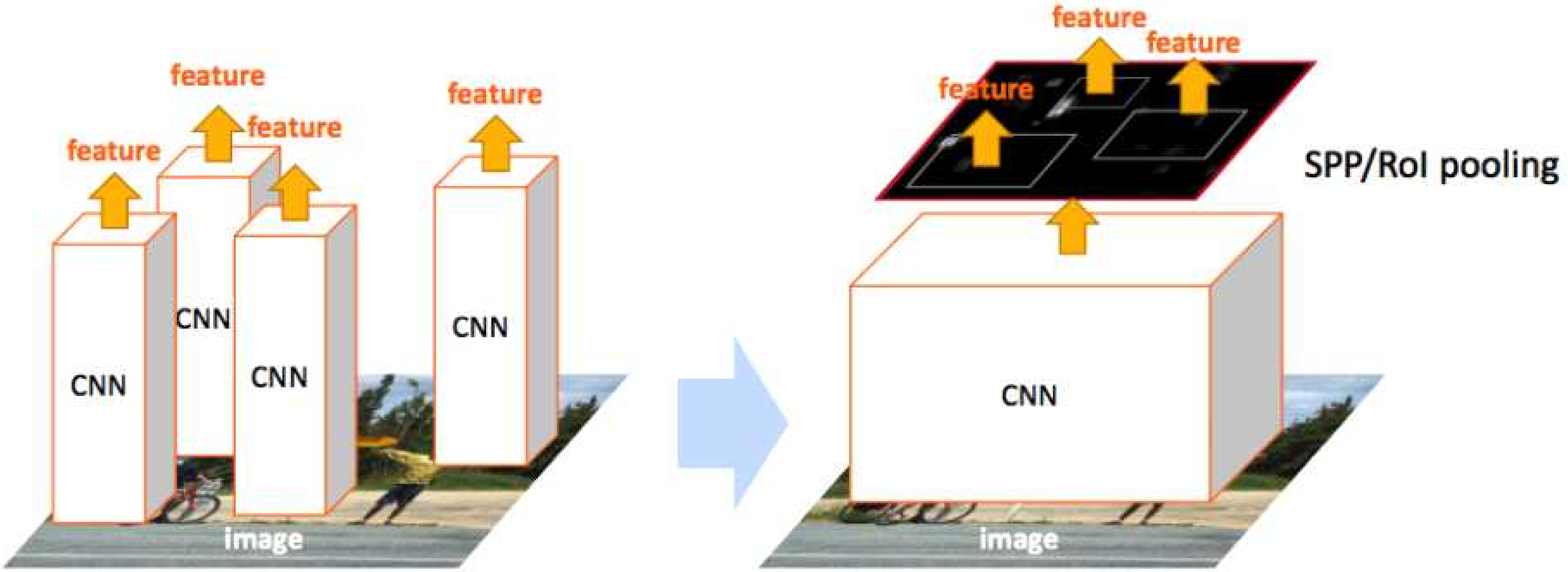
SSE(Sum of Squared Error)

$$\mathcal{L}_{reg} = \sum_{i \in \{x,y,w,h\}} \left(t_i - d_i(p)\right)^2 + \lambda \parallel w \parallel^2$$

3. Training Bounding box regressor using fine tuned AlexNet



R-CNN vs Fast R-CNN



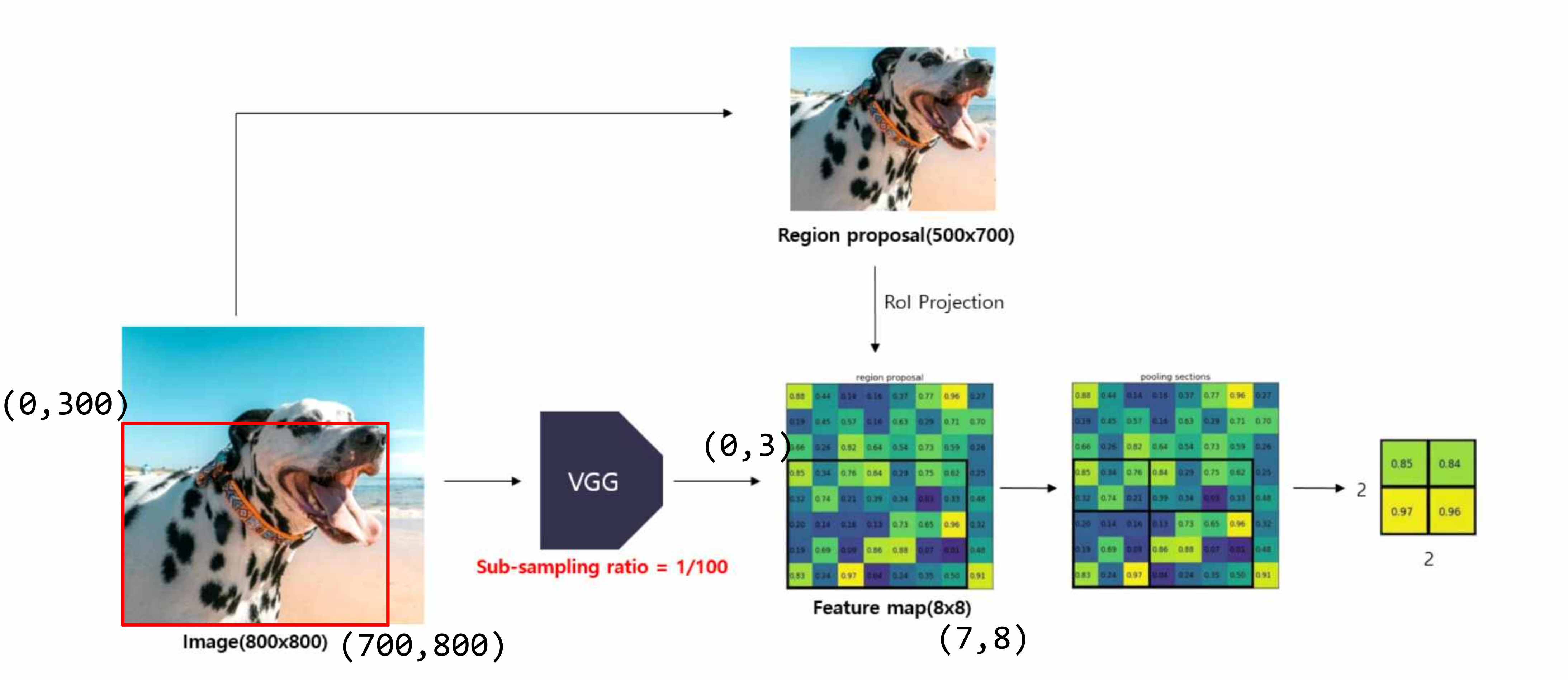
R-CNN

- Extract image regions
- 1 CNN per region (2000 CNNs)
- Classify region-based features
- Complexity: $\sim 224 \times 224 \times 2000$

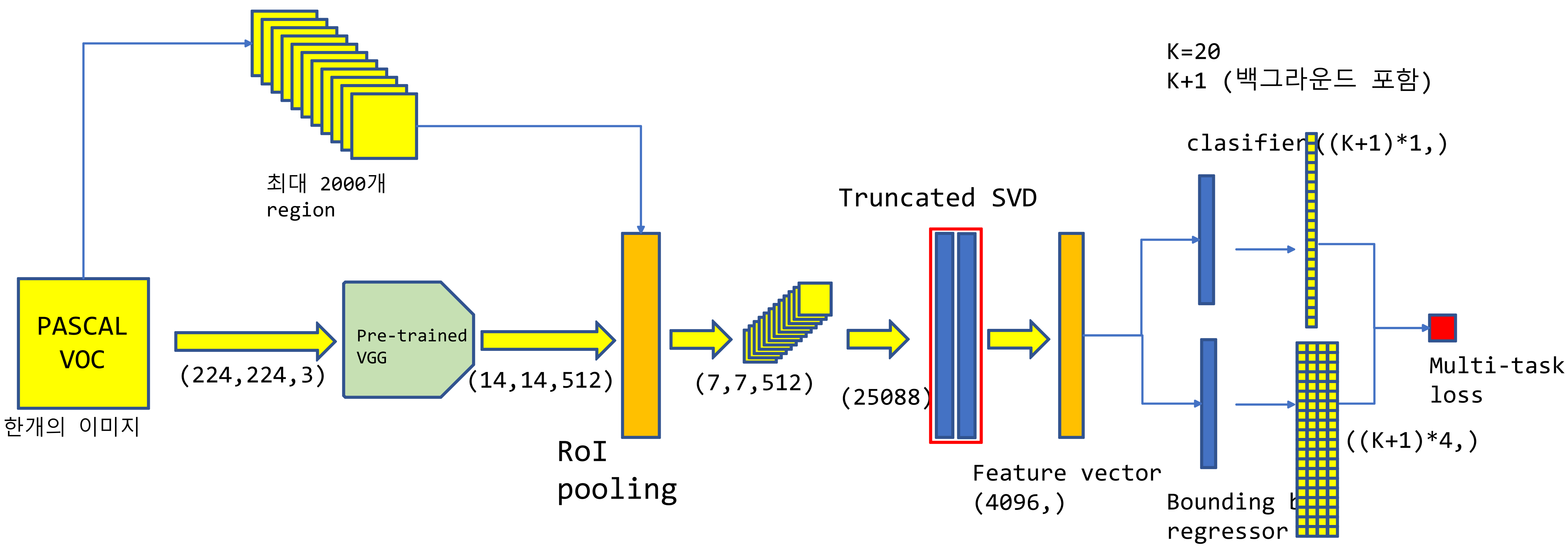
SPP-net & Fast R-CNN (the same forward pipeline)

- 1 CNN on the entire image
- Extract features from feature map regions
- Classify region-based features
- Complexity: $\sim 600 \times 1000 \times \mathbf{1}$
- ~ 160**x faster** than R-CNN

RoI pooling 개념

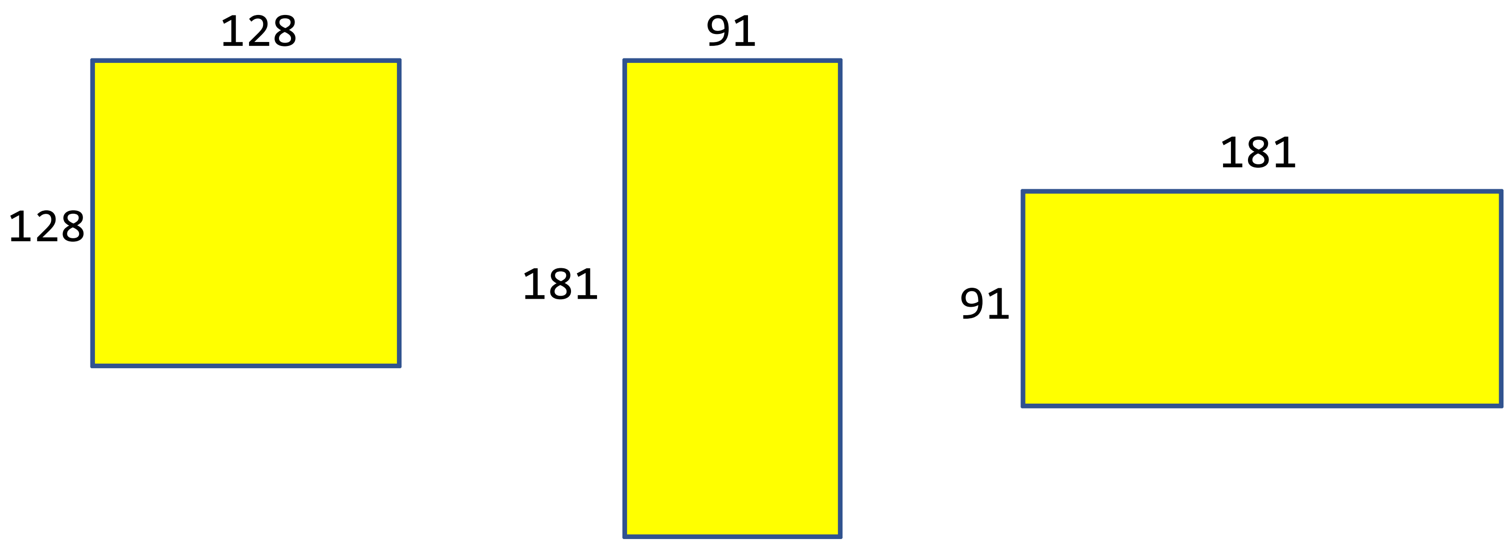


R-CNN vs Fast R-CNN



[1:1, 1:2, 2:1]

(128,128) (256,256) (512,512)



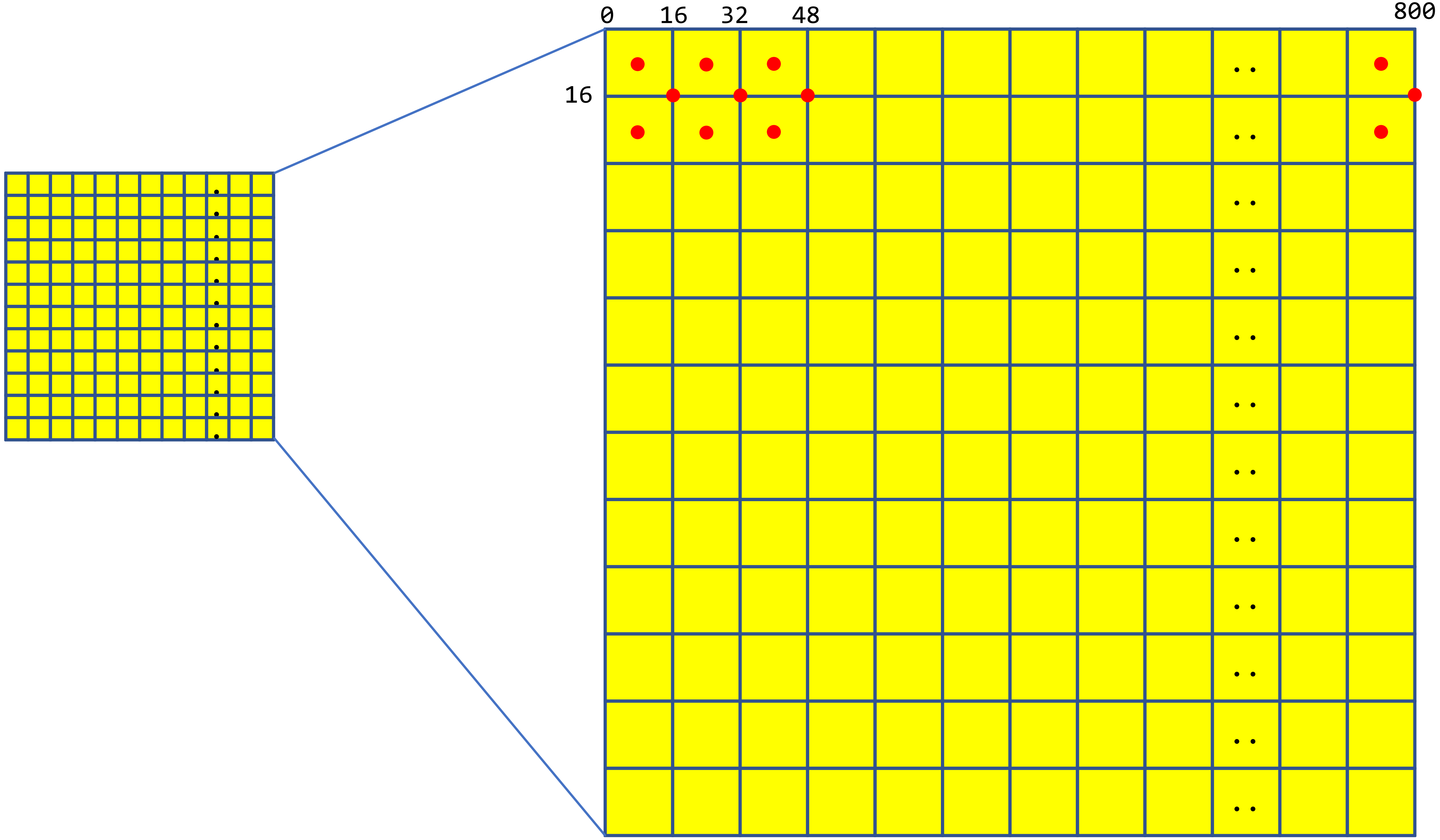
$$w \times h = s^2$$

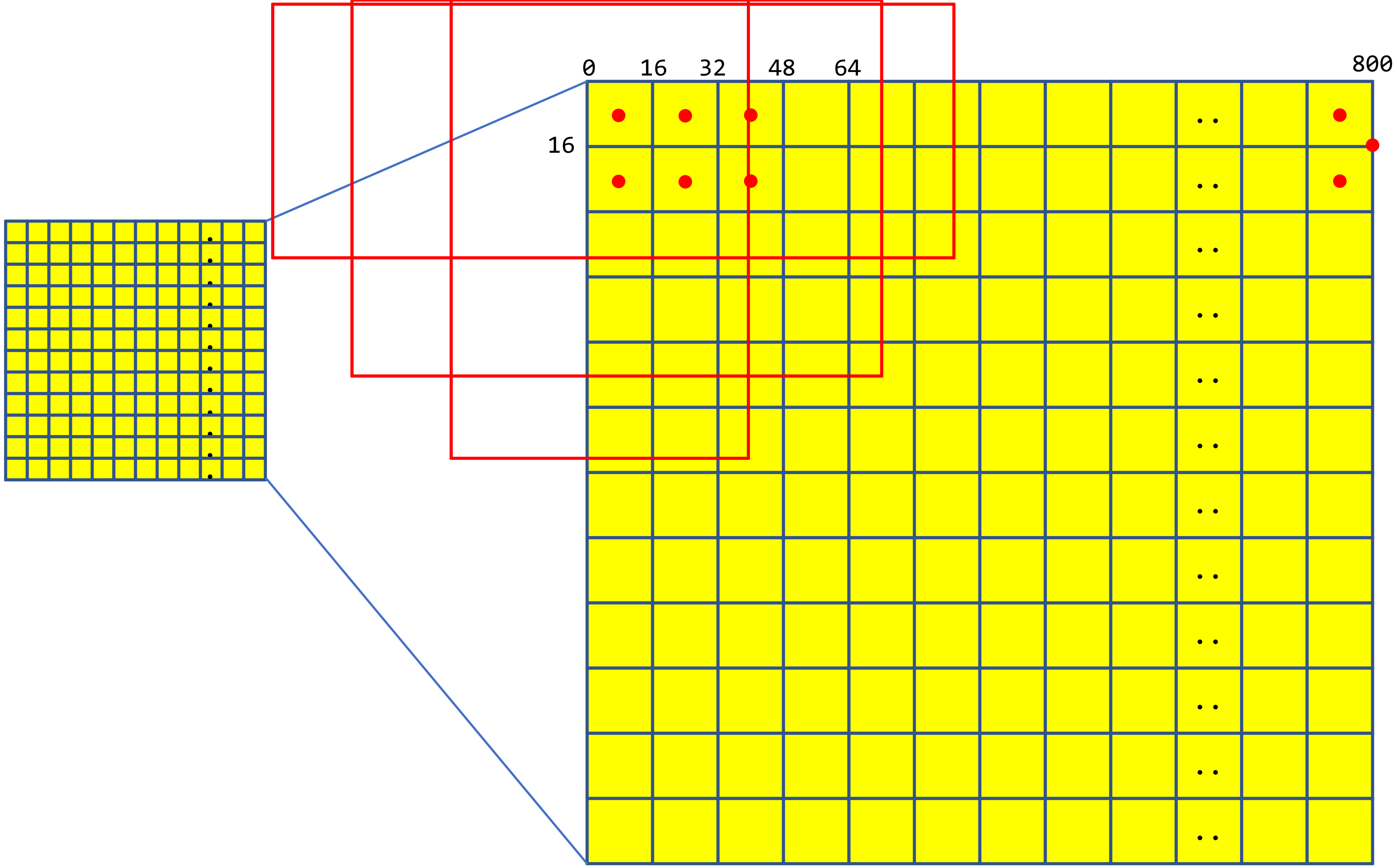
$$w = \frac{h}{2}$$

$$\frac{h^2}{2} = s^2$$

$$h = \sqrt{2 \times s^2}$$

$$w = \frac{\sqrt{2 \times s^2}}{2}$$


0
16
32
48
800
16



0
16
32
48
64
800
16

[1:1, 1:2, 2:1]

(128,128)   (256,256)   (512,512)

$$w \times h = s^2$$

```
ratios = [0.5, 1, 2]
scales = [8, 16, 32]
sub_sample = 16

anchor_boxes = np.zeros(((feature_size * feature_size * 9), 4))
index = 0

for c in ctr:                         # per anchors
    ctr_y, ctr_x = c
    for i in range(len(ratios)):      # per ratios
        for j in range(len(scales)): # per scales

            # anchor box height, width
            h = sub_sample * scales[j] * np.sqrt(ratios[i])
            w = sub_sample * scales[j] * np.sqrt(1./ ratios[i])

            # anchor box [x1, y1, x2, y2]
            anchor_boxes[index, 1] = ctr_y - h / 2.
            anchor_boxes[index, 0] = ctr_x - w / 2.
            anchor_boxes[index, 3] = ctr_y + h / 2.
            anchor_boxes[index, 2] = ctr_x + w / 2.
            index += 1
```

(50*50*9,4)

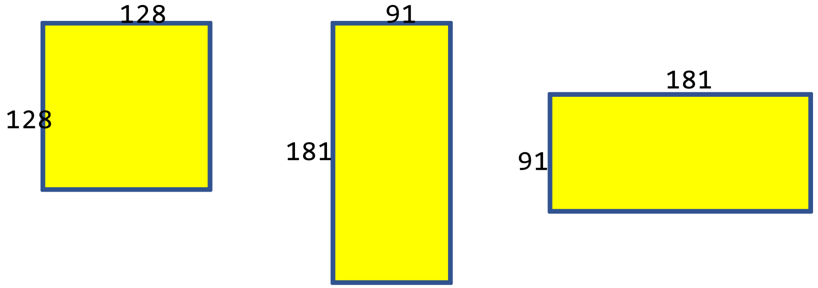
$$w = \frac{h}{2}$$

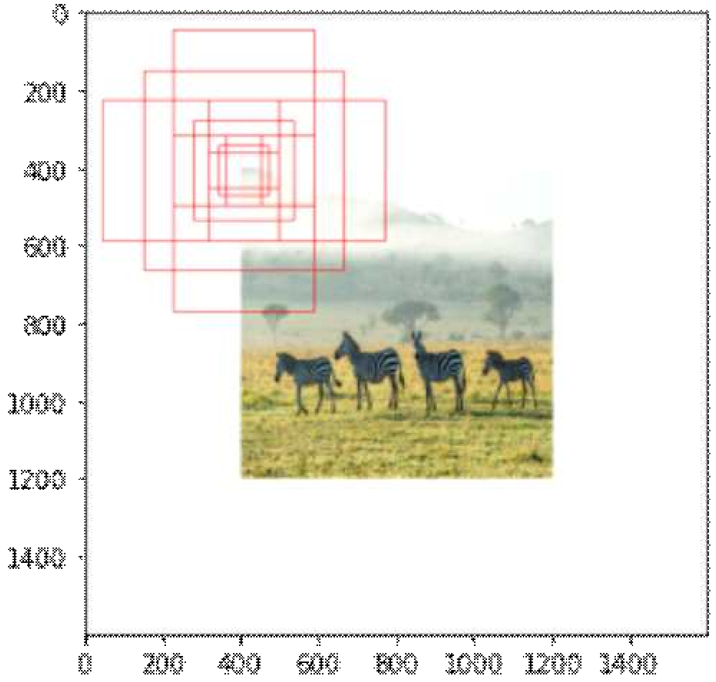
$$\frac{h^2}{2} = s^2$$

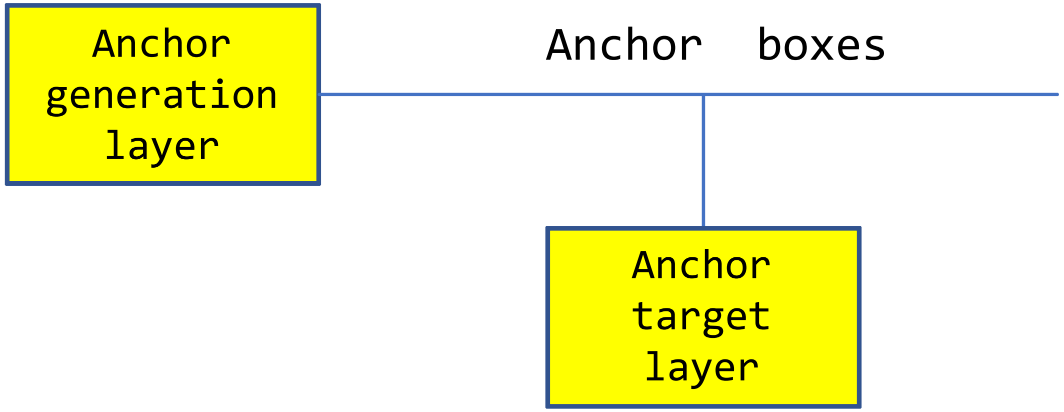
$$16 * 8 * \frac{1}{\sqrt{2}}$$

$$h = \sqrt{2 \times s^2}$$

$$w = \frac{\sqrt{2 \times s^2}}{2}$$




0
200
400
600
800
1000
1200
1400
0
200
400
600
800
1000
1200
1400




```
index_inside = np.where(
        (anchor_boxes[:, 0] >= 0) &
        (anchor_boxes[:, 1] >= 0) &
        (anchor_boxes[:, 2] <= 800) &
        (anchor_boxes[:, 3] <= 800))[0]

print(index_inside.shape)

# only 8940 anchor boxes are inside the boundary out of 22500
valid_anchor_boxes = anchor_boxes[index_inside]
print(valid_anchor_boxes.shape)
```

2) Calculate IoUs

```
ious = np.empty((len(valid_anchor_boxes),4), dtype=np.float32)
ious.fill(0)

# anchor boxes
for i, anchor_box in enumerate(valid_anchor_boxes):
    xa1, ya1, xa2, ya2 = anchor_box
    anchor_area = (xa2 - xa1) * (ya2 - ya1)

    # ground truth boxes
    for j, gt_box in enumerate(bbox):
        xb1, yb1, xb2, yb2 = gt_box
        box_area = (xb2 - xb1) * (yb2 - yb1)

        inter_x1 = max([xb1, xa1])
        inter_y1 = max([yb1, ya1])
        inter_x2 = min([xb2, xa2])
        inter_y2 = min([yb2, ya2])
...
```

2) Calculate IoUs

```
ious = np.empty((len(valid_anchor_boxes),4), dtype=np.float32)
ious.fill(0)

# anchor boxes
...
 if (inter_x1 < inter_x2) and (inter_y1 < inter_y2):
            inter_area = (inter_x2 - inter_x1) * (inter_y2 - inter_y1)
            iou = inter_area / (anchor_area + box_area - inter_area)
        else:
            iou = 0

        ious[i, j] = iou
```

3) Sample positive/negative anchor boxes

```
(8940,4)
```

```
gt_argmax_ious = ious.argmax(axis=0)
print(gt_argmax_ious)

gt_max_ious = ious[gt_argmax_ious, np.arange(ious.shape[1])]
             # ious[[1008, 2862, 5935, 8699], [0,1,2,3]]
print(gt_max_ious)

gt_argmax_ious = np.where(ious == gt_max_ious)[0]
print(gt_argmax_ious)
```

```
[[0.58514285 0.         0.         0.        ]
 [0.16435339 0.5752716  0.         0.        ]
 [0.         0.         0.5255493  0.        ]
 [0.         0.         0.         0.6325869 ]]
```

3) Sample positive/negative anchor boxes

```
...
label = np.empty((len(index_inside),), dtype=np.int32)
label.fill(-1)
print(label.shape)

pos_iou_threshold = 0.7
neg_iou_threshold = 0.3

label[gt_argmax_ious] = 1
label[max_ious >= pos_iou_threshold] = 1
label[max_ious < neg_iou_threshold] = 0
```

3) Sample positive/negative anchor boxes

```
n_sample = 256
pos_ratio = 0.5
n_pos = pos_ratio * n_sample

pos_index = np.where(label == 1)[0]

if len(pos_index) > n_pos:
    disable_index = np.random.choice(pos_index,
                                     size = (len(pos_index) - n_pos),
                                     replace=False)
    label[disable_index] = -1

n_neg = n_sample * np.sum(label == 1)
neg_index = np.where(label == 0)[0]

if len(neg_index) > n_neg:
    disable_index = np.random.choice(neg_index,
                                     size = (len(neg_index) - n_neg),
                                     replace = False)
    label[disable_index] = -1
```

```
height = valid_anchor_boxes[:, 3] - valid_anchor_boxes[:, 1]
width = valid_anchor_boxes[:, 2] - valid_anchor_boxes[:, 0]
ctr_y = valid_anchor_boxes[:, 1] + 0.5 * height
ctr_x = valid_anchor_boxes[:, 0] + 0.5 * width

base_height = max_iou_bbox[:, 3] - max_iou_bbox[:, 1]
base_width = max_iou_bbox[:, 2] - max_iou_bbox[:, 0]
base_ctr_y = max_iou_bbox[:, 1] + 0.5 * base_height
base_ctr_x = max_iou_bbox[:, 0] + 0.5 * base_width

eps = np.finfo(height.dtype).eps
print(eps)
print(height.shape, width.shape)
height = np.maximum(height, eps)
width = np.maximum(width, eps)

print(height[1000:1100])
print(height.shape, width.shape)

dy = (base_ctr_y - ctr_y) / height
dx = (base_ctr_x - ctr_x) / width
dh = np.log(base_height / height)
dw = np.log(base_width / width)
```

$$t_x = (g_x - p_x)/p_w$$

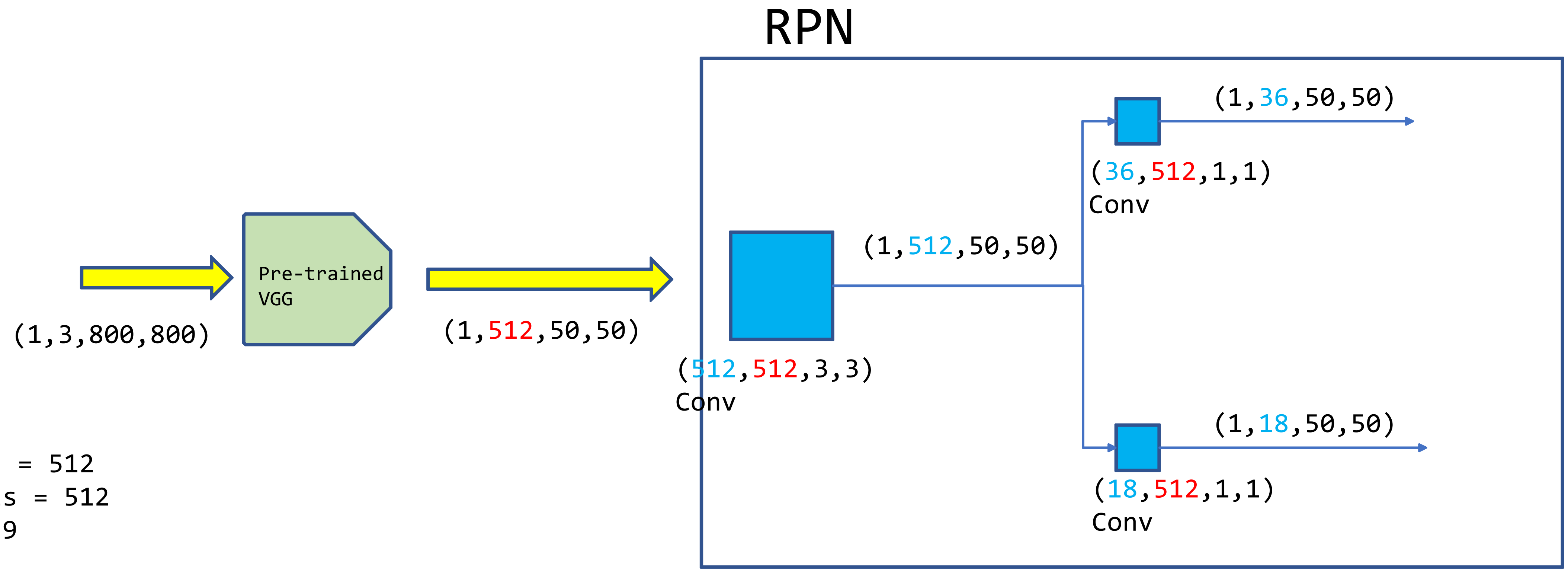
$$t_y = (g_y - p_y)/p_h$$

$$t_w = \log(g_w/p_w)$$

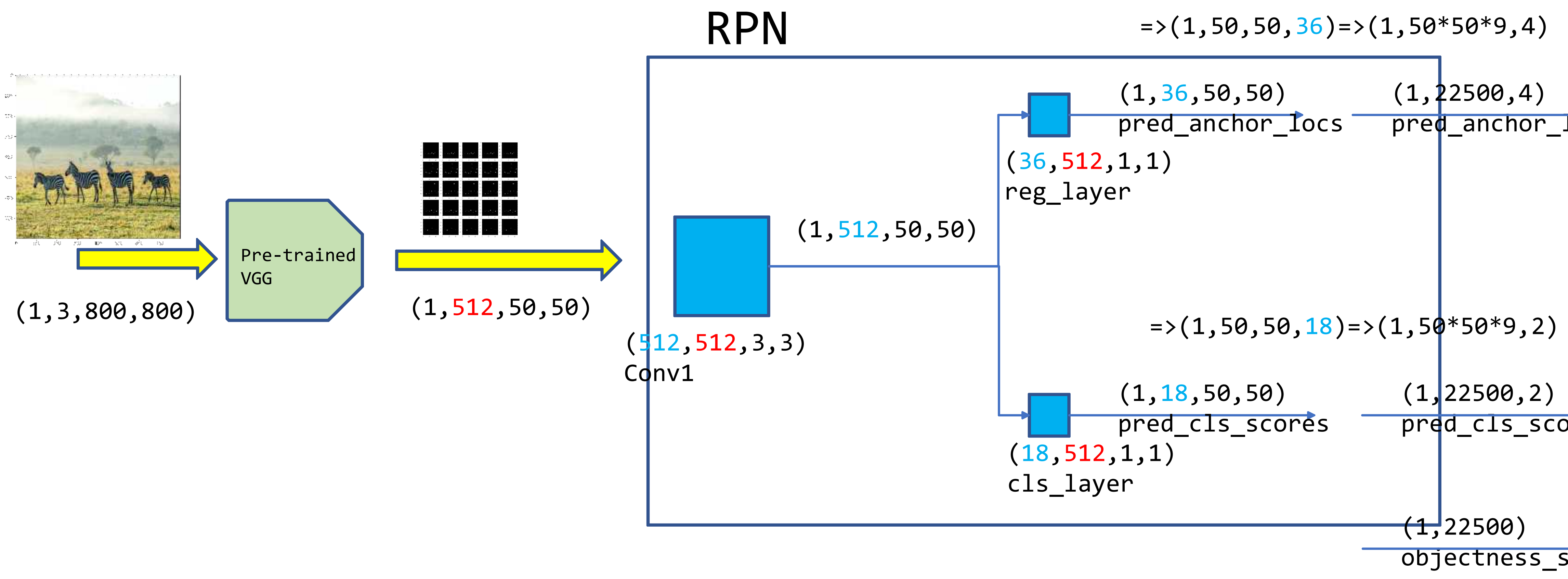
$$t_h = \log(g_h/h)$$

anchor_locs

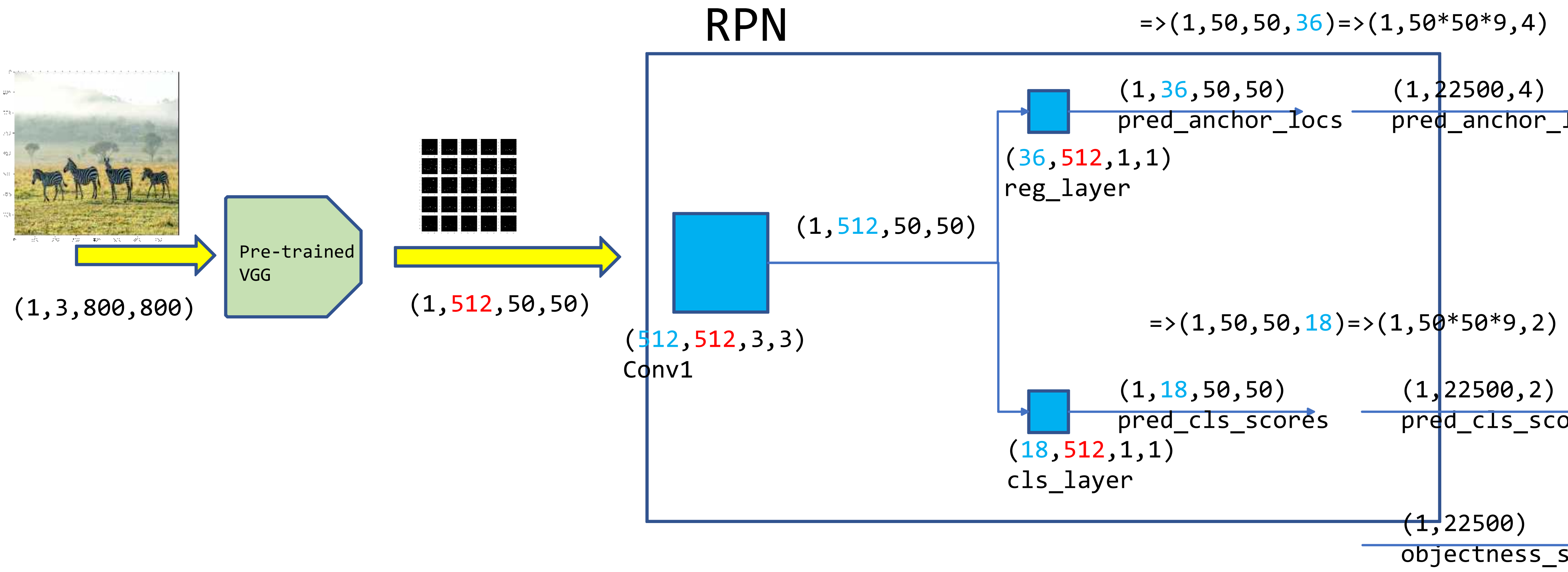
```
[[ 1.24848541  2.49973296  0.56971714 -0.03
 [ 1.24848541  2.41134461  0.56971714 -0.03
 [ 1.24848541  2.32295626  0.56971714 -0.03
 ...
 [-0.5855728  -0.63252911  0.4917556  -0.03
 [-0.5855728  -0.72091746  0.4917556  -0.03
 [-0.5855728  -0.80930581  0.4917556  -0.03
```




```
in_channels = 512
mid_channels = 512
n_anchor = 9

conv1 = nn.Conv2d(in_channels, mid_channels, 3, 1, 1).to(DEVICE)
conv1.weight.data.normal_(0, 0.01)
conv1.bias.data.zero_()

# bounding box regressor
reg_layer = nn.Conv2d(mid_channels, n_anchor * 4, 1, 1, 0).to(DEVICE)
reg_layer.weight.data.normal_(0, 0.01)
reg_layer.bias.data.zero_()

# classifier(object or not)
cls_layer = nn.Conv2d(mid_channels, n_anchor * 2, 1, 1, 0).to(DEVICE)
cls_layer.weight.data.normal_(0, 0.01)
cls_layer.bias.data.zero_()
```




```
x = conv1(output_map.to(DEVICE)) # output_map = faster_rcnn_feature_extractor(imgTensor)
pred_anchor_locs = reg_layer(x) # bounding box regresor output
pred_cls_scores = cls_layer(x)  # classifier output

print(pred_anchor_locs.shape, pred_cls_scores.shape)
```

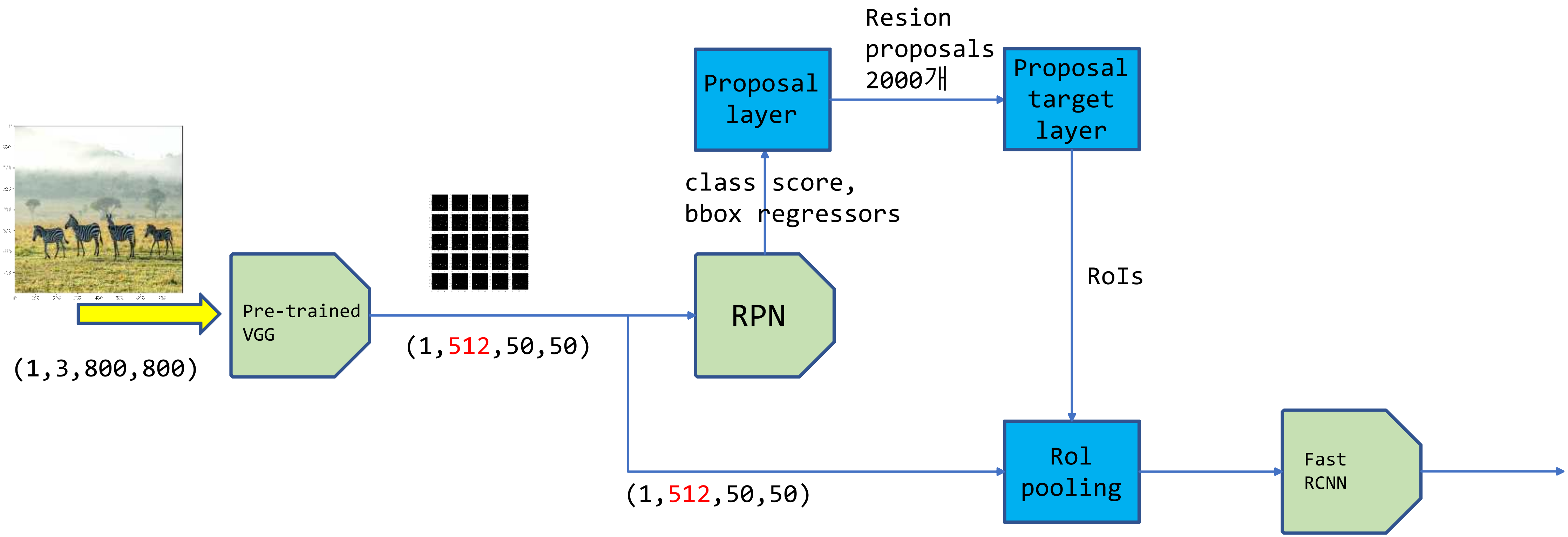



```
rpn_cls_loss = F.cross_entropy(rpn_score, gt_rpn_score.long().to(DEVICE), ignore_index = -1)
x = torch.abs(mask_loc_targets.cpu() - mask_loc_preds.cpu())
rpn_loc_loss = ((x < 1).float() * 0.5 * x ** 2) + ((x >= 1).float() * (x - 0.5))
rpn_lambda = 10
N_reg = (gt_rpn_score > 0).float().sum()
rpn_loc_loss = rpn_loc_loss.sum() / N_reg
rpn_loss = rpn_cls_loss + (rpn_lambda * rpn_loc_loss)
```

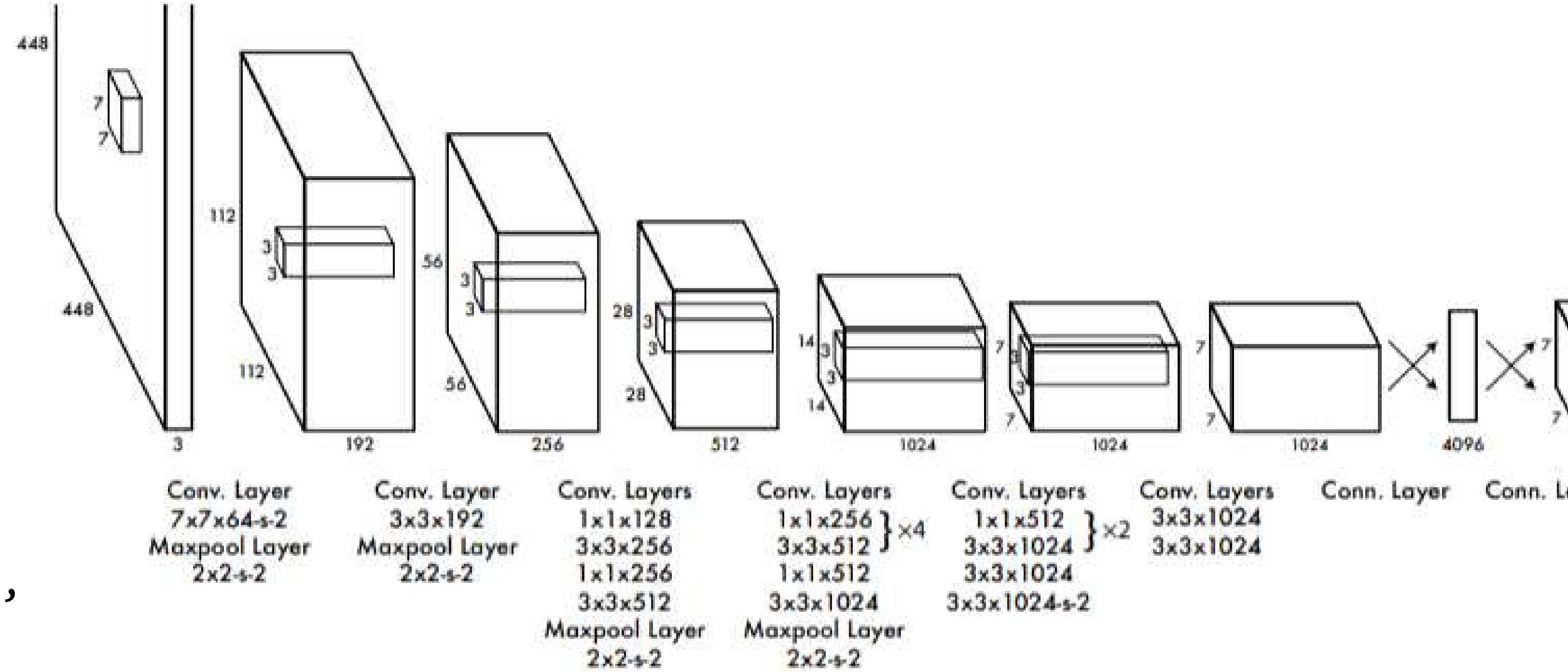
$$L(\{p_i\},\{t_i\}) = \frac{1}{N_{cls}}\sum_i L_{cls}(p_i, p_i^*) + \lambda\frac{1}{N_{reg}}\sum_i p_i^* L_{reg}(t_i, t_i^*)$$

- i : mini-batch 내의 anchor의 index
- p_i : anchor i에 객체가 포함되어 있을 예측 확률
- p_i^* : anchor가 양성일 경우 1, 음성일 경우 0을 나타내는 index parameter
- t_i : 예측 bounding box의 파라미터화된 좌표(coefficient)
- t_i^* : ground truth box의 파라미터화된 좌표
- L_{cls} : Loss loss
- L_{reg} : Smooth L1 loss
- N_{cls} : mini-batch의 크기(논문에서는 256으로 지정)
- N_{reg} : anchor 위치의 수
- λ : balancing parameter(default=10)

```
rpn_cls_loss = F.cross_entropy(rpn_score, gt_rpn_score.long().to(DEVICE), ignore_index = -1)
x = torch.abs(mask_loc_targets.cpu() - mask_loc_preds.cpu())
rpn_loc_loss = ((x < 1).float() * 0.5 * x ** 2) + ((x >= 1).float() * (x - 0.5))
rpn_lambda = 10
N_reg = (gt_rpn_score > 0).float().sum()
rpn_loc_loss = rpn_loc_loss.sum() / N_reg
rpn_loss = rpn_cls_loss + (rpn_lambda * rpn_loc_loss)
```


Resion
proposals
2000개
Proposal
layer
Proposal
target
layer
class score,
bbox regressors
RoIs
Pre-trained
VGG
RPN
(1,3,800,800)
(1,512,50,50)
(1,512,50,50)
RoI
pooling
Fast
RCNN

```
architecture_config = [
    (7, 64, 2, 3),  => (1,64,224,224)
    "M",            => (1,64,112,112)
    (3, 192, 1, 1),
    "M",
    (1, 128, 1, 0),
    (3, 256, 1, 1),
    (1, 256, 1, 0),
    (3, 512, 1, 1),
    "M",
    [(1, 256, 1, 0), (3, 512, 1, 1), 4],
    (1, 512, 1, 0),
    (3, 1024, 1, 1),
    "M",
    [(1, 512, 1, 0), (3, 1024, 1, 1), 2],
    (3, 1024, 1, 1),
    (3, 1024, 2, 1),
    (3, 1024, 1, 1),
    (3, 1024, 1, 1),
]
```



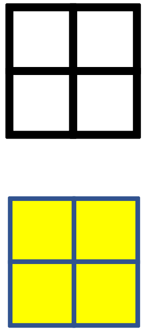
(1,3,448,448)

N=448
F=7
S=2
P=3
FN=64

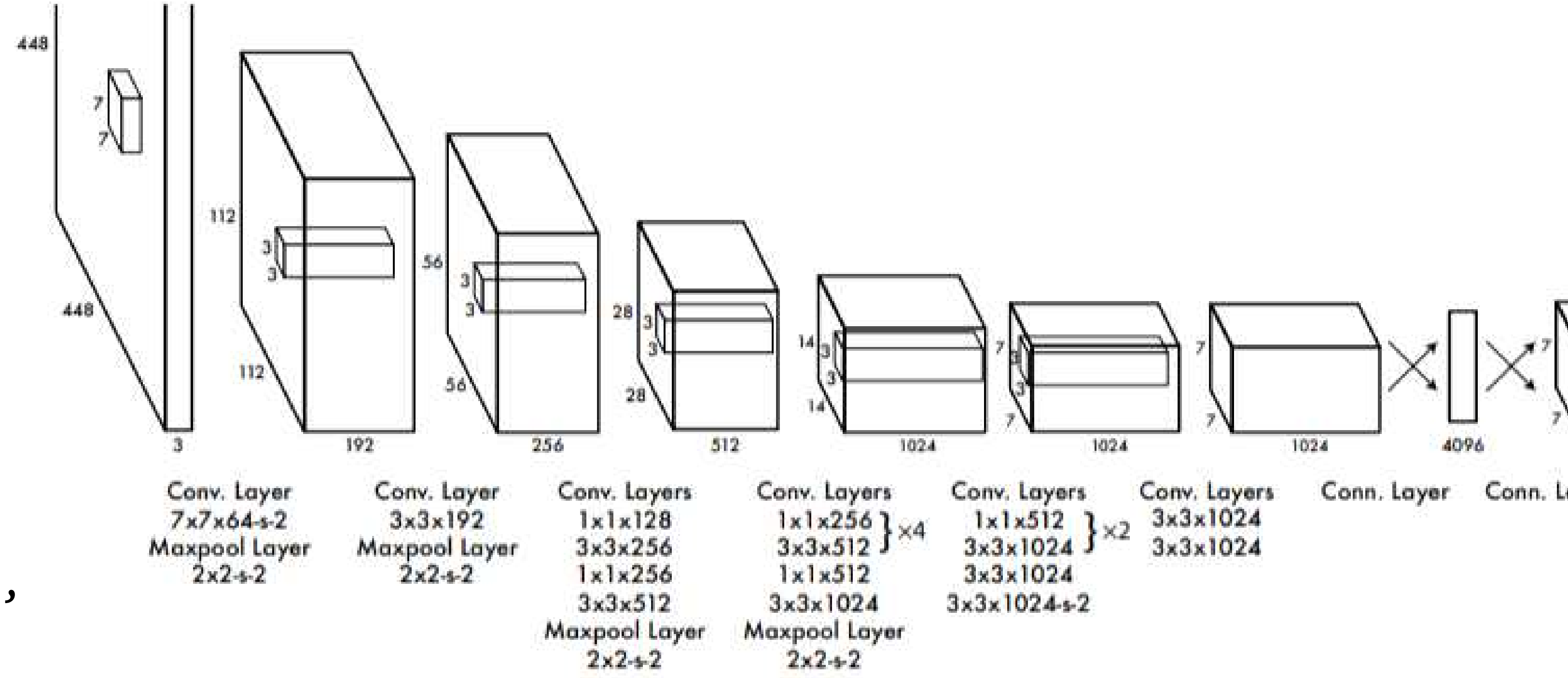
(1,64,224,224)

$$\frac{N-F+1+2P}{S} \qquad \frac{448-7+1+3*2}{2}$$

0	0	0	0	0	0
0	1	2	3	4	0
0	5	6	7	8	0
0	9	10	11	12	0
0	13	14	15	16	0
0	0	0	0	0	0



```
architecture_config = [
    (7, 64, 2, 3), => (1,64,224,224)
    "M",           => (1,64,112,112)
    (3, 192, 1, 1),
    "M",           => (1,192,56,56)
    (1, 128, 1, 0),
    (3, 256, 1, 1),
    (1, 256, 1, 0),
    (3, 512, 1, 1),
    "M",
    [(1, 256, 1, 0), (3, 512, 1, 1), 4],
    (1, 512, 1, 0),
    (3, 1024, 1, 1),
    "M",
    [(1, 512, 1, 0), (3, 1024, 1, 1), 2],
    (3, 1024, 1, 1),
    (3, 1024, 2, 1),
    (3, 1024, 1, 1),
    (3, 1024, 1, 1),
]
```



(1,64,112,112)

F=3 (1,192,112,112)

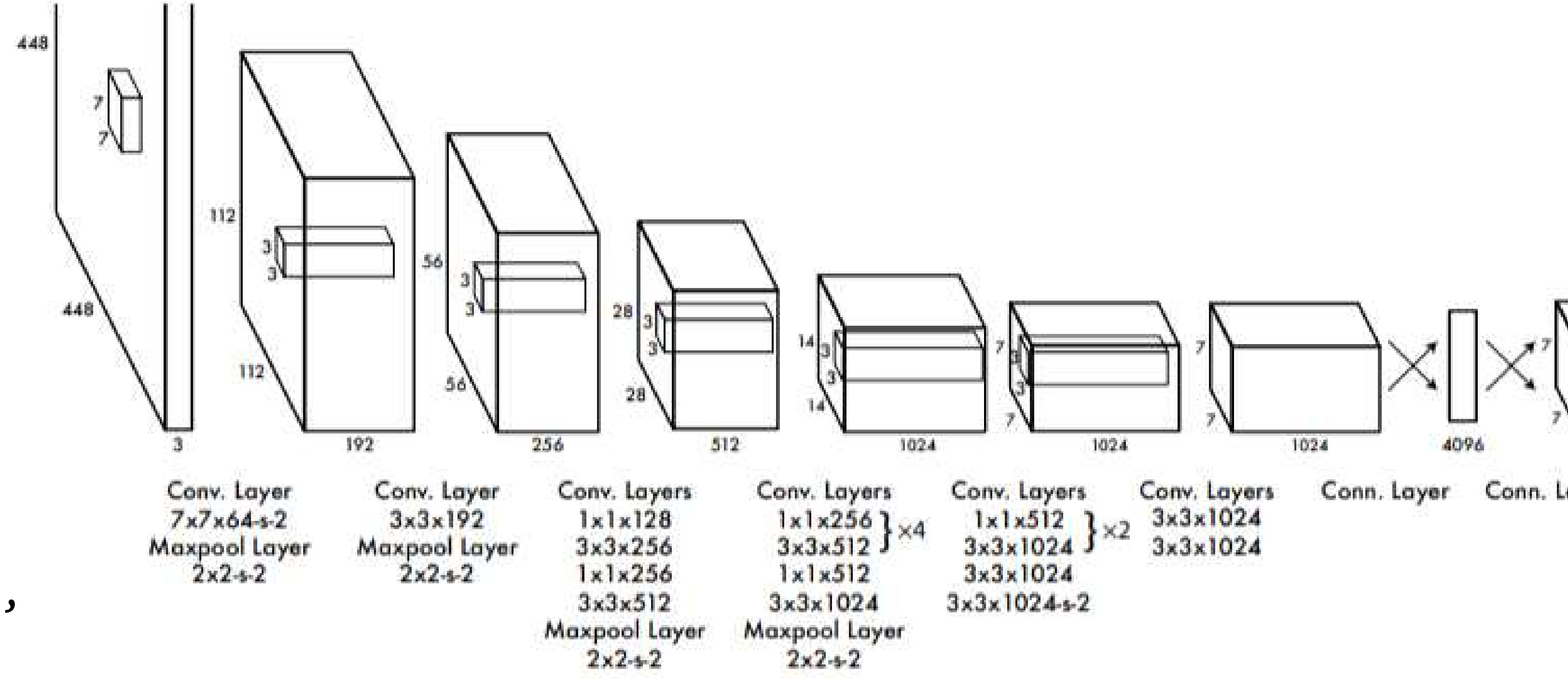
S=1

P=1

FN=192

$$\frac{N-F+1+2P}{S} \qquad \frac{112-3+1+1*2}{1}$$

```
architecture_config = [
    (7, 64, 2, 3),  => (1,64,224,224)
    "M",            => (1,64,112,112)
    (3, 192, 1, 1),
    "M",            => (1,192,56,56)
    (1, 128, 1, 0), => (1,128,56,56)
    (3, 256, 1, 1), => (1,256,56,56)
    (1, 256, 1, 0), => (1,256,56,56)
    (3, 512, 1, 1), => (1,512,56,56)
    "M",            => (1,512,28,28)
    [(1, 256, 1, 0), (3, 512, 1, 1), 4],
    (1, 512, 1, 0),
    (3, 1024, 1, 1),=> (1,1024,28,28)
    "M",            => (1,1024,14,14)
    [(1, 512, 1, 0), (3, 1024, 1, 1), 2],
    (3, 1024, 1, 1),
    (3, 1024, 2, 1),=> (1,1024,7,7)
    (3, 1024, 1, 1),
    (3, 1024, 1, 1),
]
```



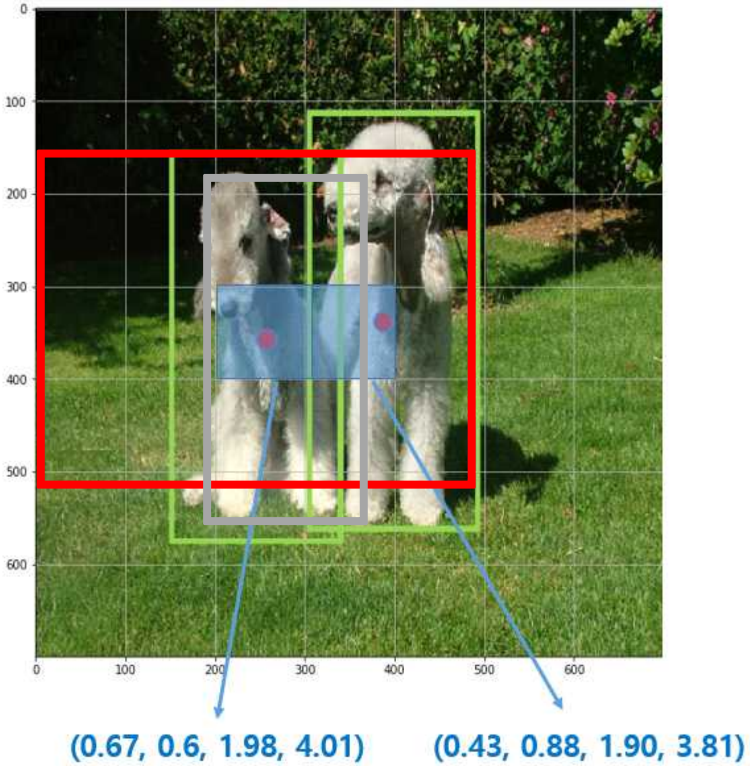
(1,192,56,56)

F=1
S=1
P=0
FN=128

(1,128,56,56)

$$\frac{N-F+1+2P}{S} \qquad \frac{56-1+1+0*2}{1}$$

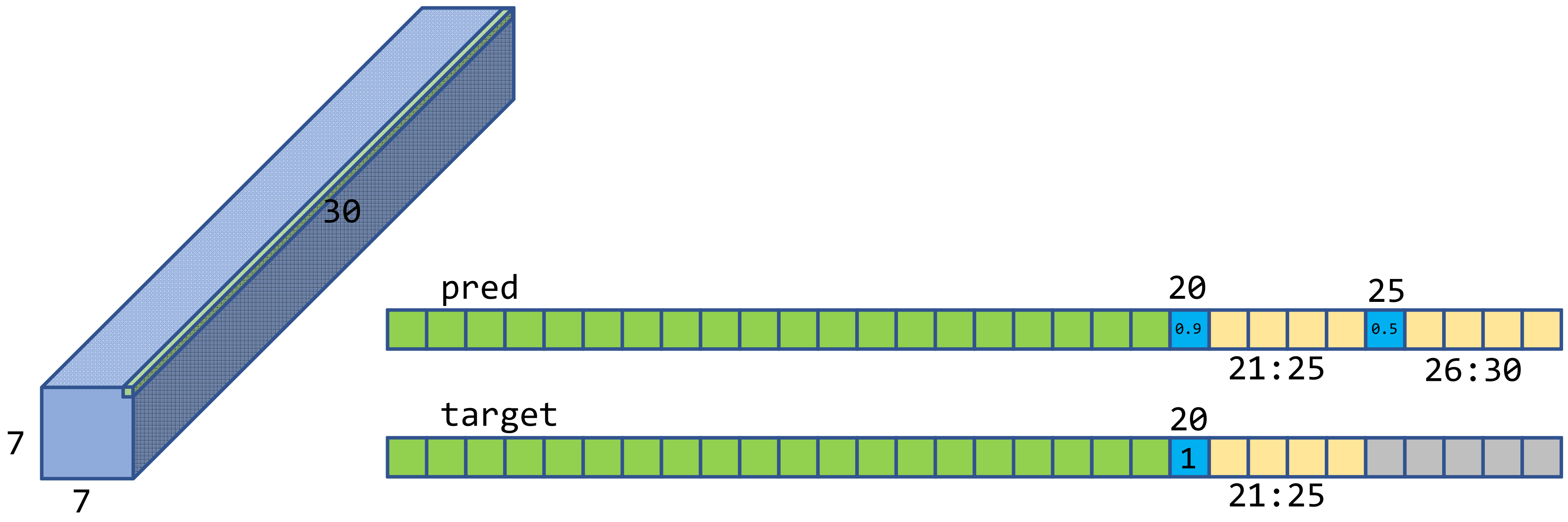
BBox 중심 x, y좌표Loss

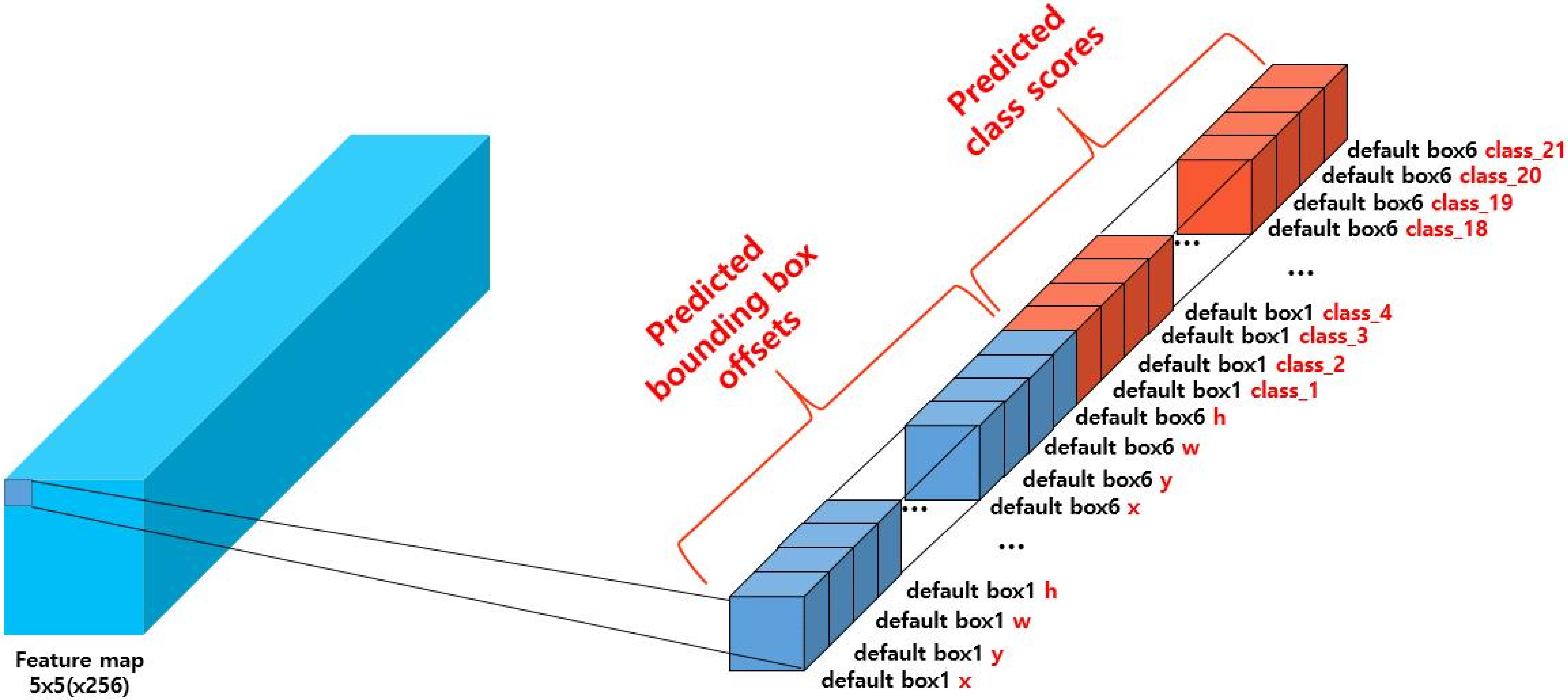


$$\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} \left((x - \hat{x}_i)^2 + (y - \hat{y}_i)^2\right) + \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} \left(\sqrt{w} - \sqrt{\hat{w}_i}\right)^2 + \left(\sqrt{h} - \sqrt{\hat{h}\right.$$

예측 좌표 x, y 값과 Ground Truth x, y값의 오차 제곱을 기반
모든 Cell의 2개의 Bbox(89개 Bbox)중에 예측 Bbox를 책임지는 Bbox만 Loss 계산
$\mathbb{1}_{ij}^{obj}$는 98개의 Bbox 중 오브젝트 예측을 책임지는 Bbox만 1, 나머지는 0

- λ_{coord} : 많은 grid cell은 객체를 포함하지 않아 confidence score가 0이 되어 객체를 포함하는 grid cell의 gradient를 압도 모델이 불안정해질 수 있습니다. λ_{coord}는 이러한 문제를 해결하기 위해 객체를 포함하는 cell에 가중치를 두는 파라미터입니다 문에서는 $\lambda_{coord} = 5$로 설정합니다.
- S^2 : grid cell의 수(=7x7=49)
- B : grid cell별 bounding box의 수(=2)
- $1_{i,j}^{obj}$: i번째 grid cell의 j번째 bounding box가 객체를 예측하도록 할당(responsible for)받았을 때 1, 그렇지 않을 경우 0인 index parameter입니다. 앞서 설명했듯이 grid cell에서는 B개의 bounding box를 예측하지만 그 중 confidence score가 오직 1개의 bounding box만을 학습에 사용합니다.
- x_i, y_i, w_i, h_i : ground truth box의 x, y 좌표와 width, height. 여기서 크기가 큰 bounding box의 작은 오류가 크기가 작은 bounding box의 오류보다 덜 중요하다는 것을 반영하기 위해 w_i, h_i 값에 루트를 씌어주게 됩니다.
- $\hat{x}_i, \hat{y}_i, \hat{w}_i, \hat{h}_i$: 예측 bounding box의 x, y 좌표, width, height


30
7
7
pred
20
25
0.9
0.5
21:25
26:30
target
20
1
21:25


Feature map
5x5(x256)
Predicted bounding box offsets
Predicted class scores
default box1 x
default box1 y
default box1 w
default box1 h
...
default box6 x
default box6 y
default box6 w
default box6 h
default box1 class_1
default box1 class_2
default box1 class_3
default box1 class_4
...
default box6 class_18
default box6 class_19
default box6 class_20
default box6 class_21

```python
def call(self, y_true, y_pred):
        cross_entropy = tf.nn.sigmoid_cross_entropy_with_logits(
            labels=y_true, logits=y_pred
        )
        probs = tf.nn.sigmoid(y_pred)
        alpha = tf.where(tf.equal(y_true, 1.0), self._alpha, (1.0 - self._alpha))
        pt = tf.where(tf.equal(y_true, 1.0), probs, 1 - probs)
        loss = alpha * tf.pow(1.0 - pt, self._gamma) * cross_entropy
        return tf.reduce_sum(loss, axis=-1)
```

Focal Loss (if y_true == 1:)

$$C(p, y) = -\alpha \sum_i y_i (1 - p_i)^\gamma \ln(p_i)$$

Focal Loss(if y_true == 0:)

$$C(p, y) = -(1 - \alpha) \sum_i y_i (1 - p_i)^\gamma \ln(p_i)$$

```python
def call(self, y_true, y_pred):
        cross_entropy = tf.nn.sigmoid_cross_entropy_with_logits(
            labels=y_true, logits=y_pred
        )
        probs = tf.nn.sigmoid(y_pred)
        alpha = tf.where(tf.equal(y_true, 1.0), self._alpha, (1.0 - self._alpha))
        pt = tf.where(tf.equal(y_true, 1.0), probs, 1 - probs)
        loss = alpha * tf.pow(1.0 - pt, self._gamma) * cross_entropy
        return tf.reduce_sum(loss, axis=-1)
```

Focal Loss (if y_true == 1:)

$$C(p, y) = -\alpha \sum_i y_i (1 - p_i)^{\gamma} \ln(p_i)$$

Focal Loss(if y_true == 0:)

$$C(p, y) = -(1 - \alpha) \sum_i y_i (1 - p_i)^{\gamma} \ln(p_i)$$

```
f call(self, images, training=False):
    c3_output, c4_output, c5_output = self.backbone(images, training=training)
    p3_output = self.conv_c3_1x1(c3_output)
    p4_output = self.conv_c4_1x1(c4_output)
    p5_output = self.conv_c5_1x1(c5_output)
    p4_output = p4_output + self.upsample_2x(p5_output)
    p3_output = p3_output + self.upsample_2x(p4_output)
    p3_output = self.conv_c3_3x3(p3_output)
    p4_output = self.conv_c4_3x3(p4_output)
    p5_output = self.conv_c5_3x3(p5_output)
    p6_output = self.conv_c6_3x3(c5_output)
    p7_output = self.conv_c7_3x3(tf.nn.relu(p6_output))
    return p3_output, p4_output, p5_output, p6_output, p7_output
```