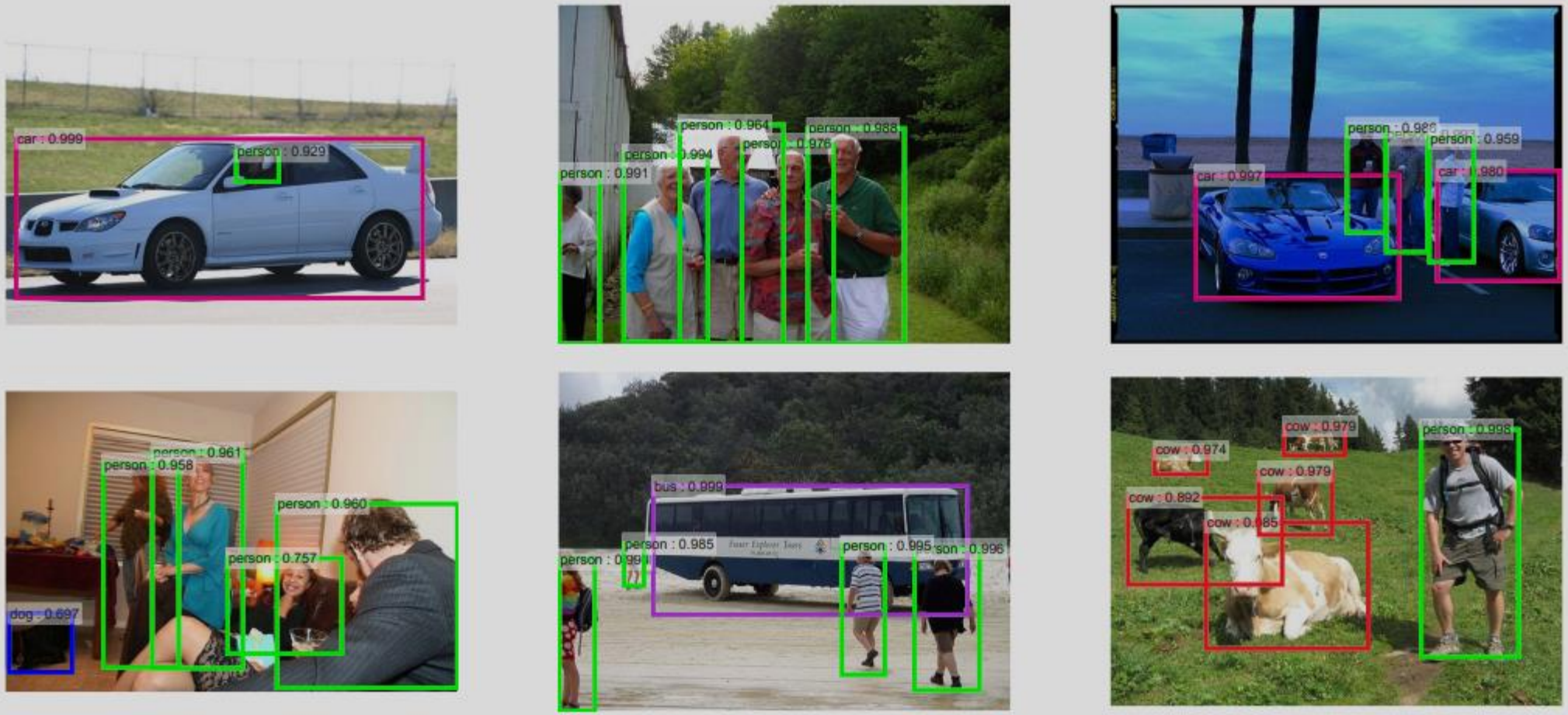
Computer Vision

Lecture 06: Computer Vision Applications

Computer vision applications

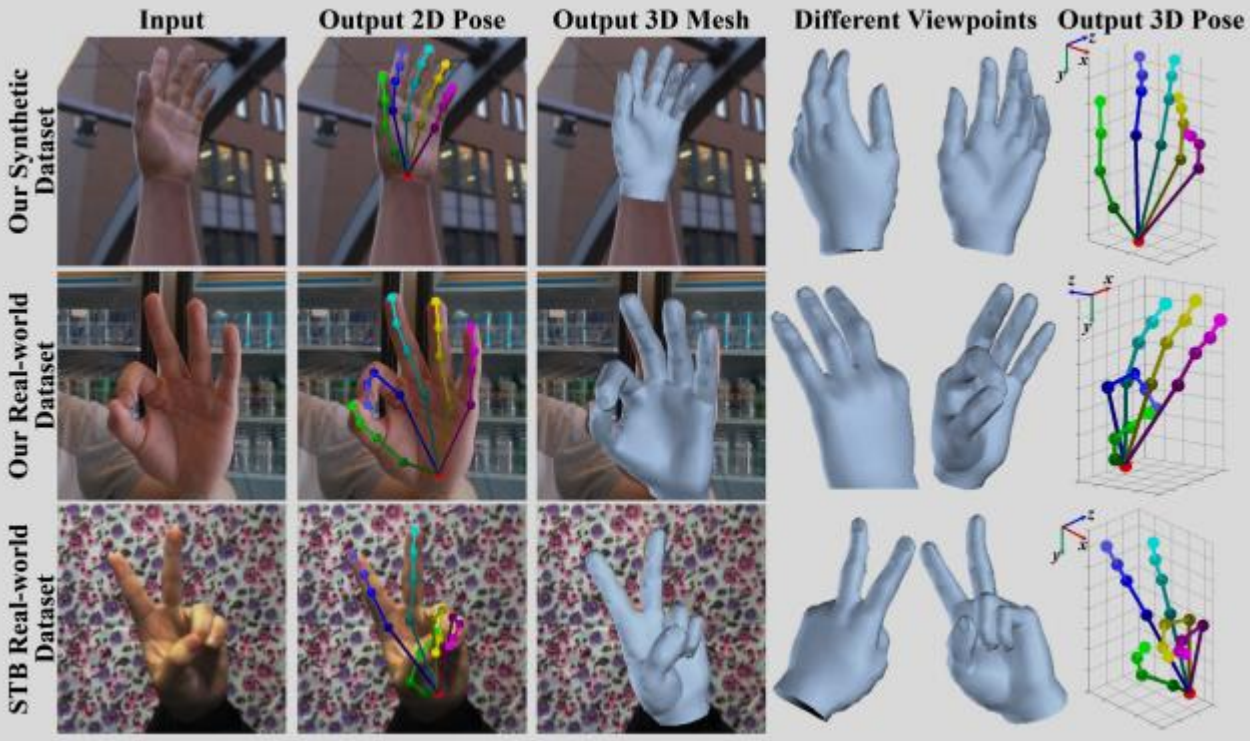


Detecting object locations. [Faster-RCNN NIPS'15]

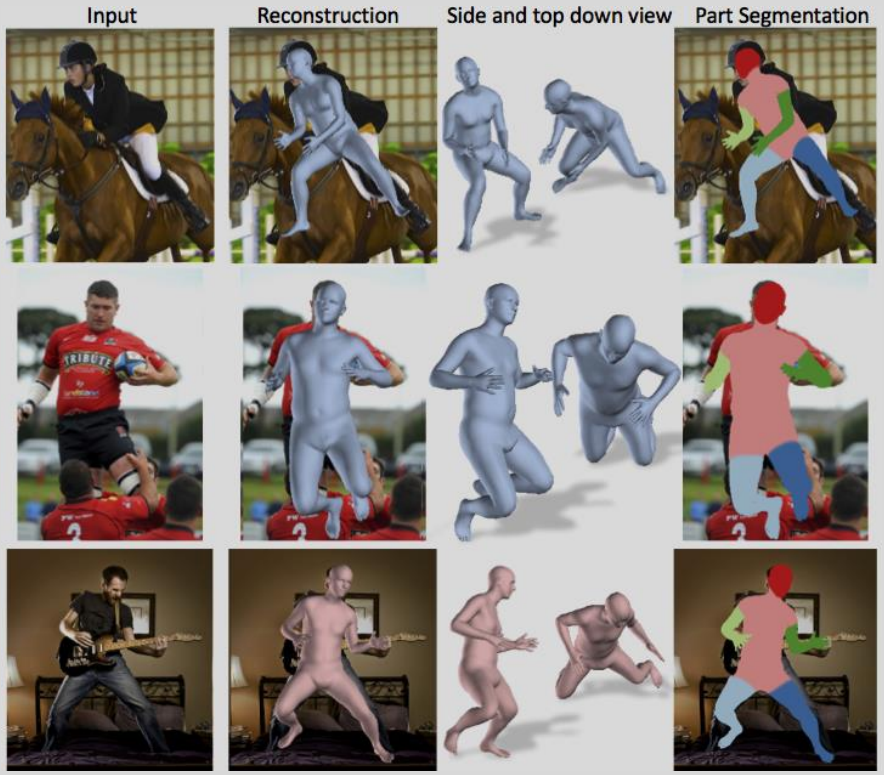
Computer vision applications

Detecting object locations and segmentation. [Mask RCNN ICCV'17]

Computer vision applications

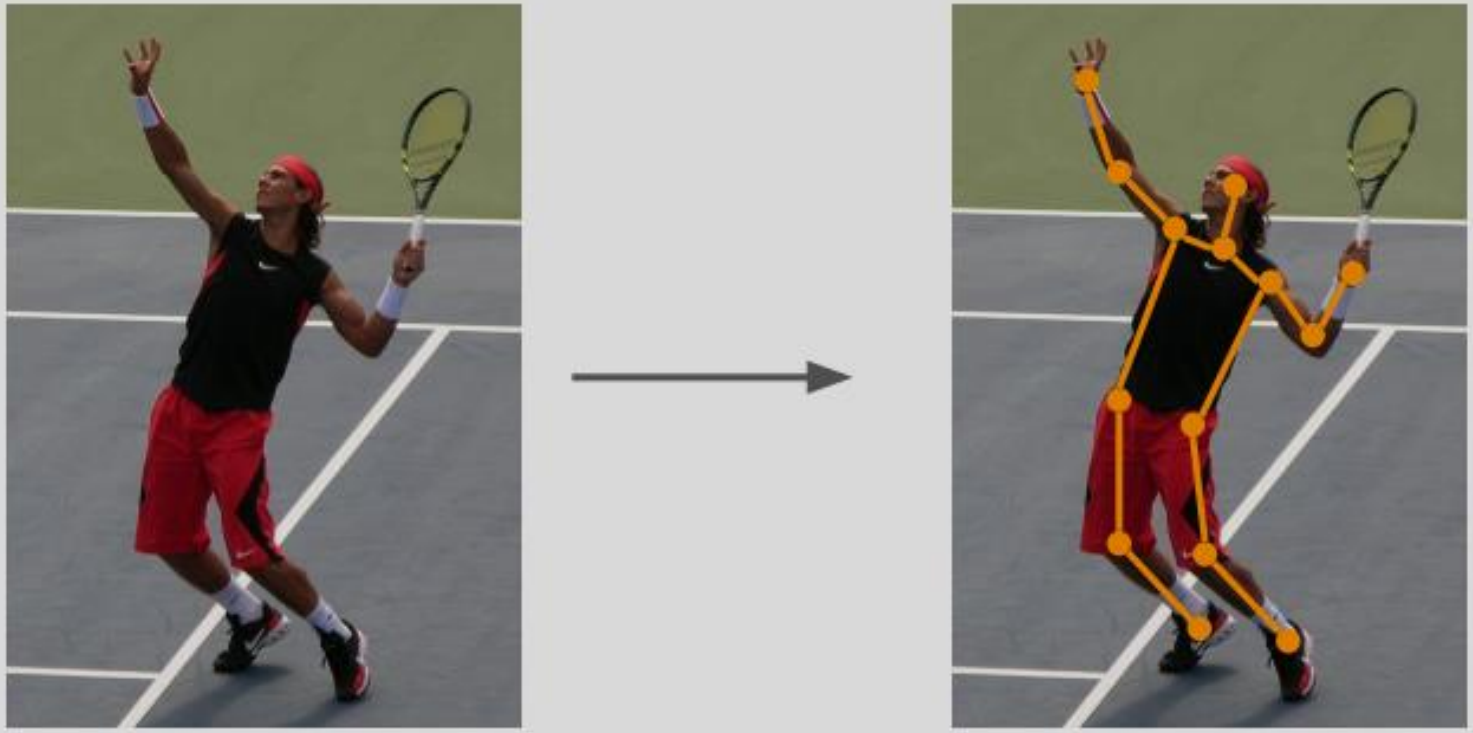


3D hand mesh reconstruction (Ge et al. CVPR'19)



3D human mesh reconstruction (Kanazawa et al. CVPR'18)

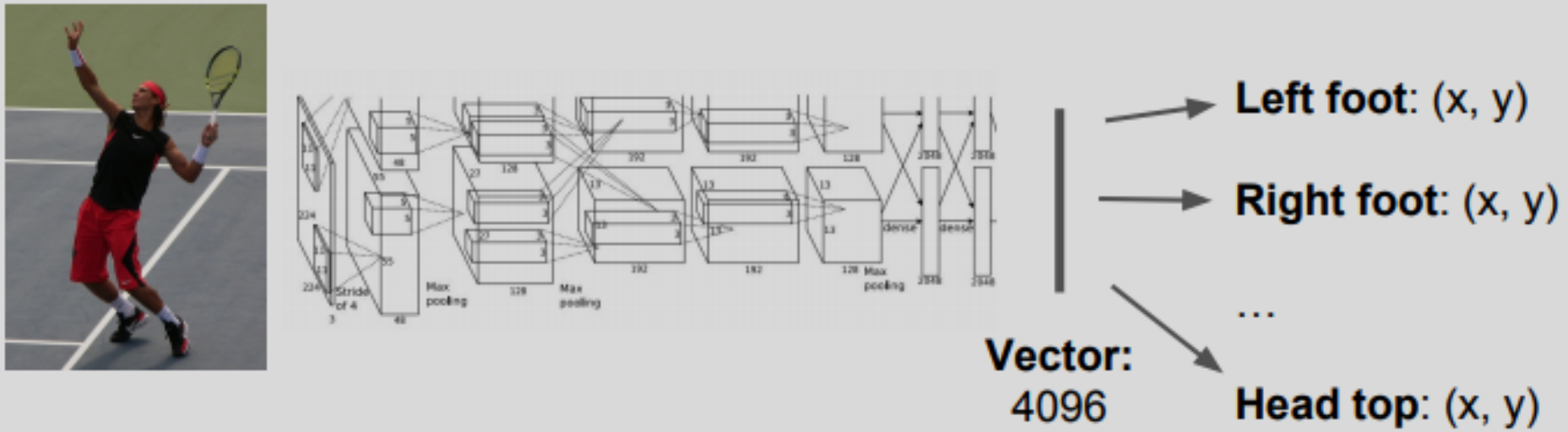
Pose estimation



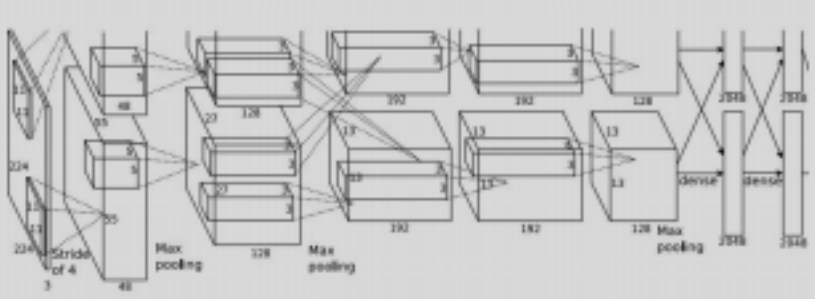
Represent pose as a set of 14 joint positions:

Left / right foot
Left / right knee
Left / right hip
Left / right shoulder
Left / right elbow
Left / right hand
Neck
Head top

Pose estimation



Pose estimation



Vector:
4096

Left foot: (x, y) → L2 loss

Right foot: (x, y) → L2 loss

…

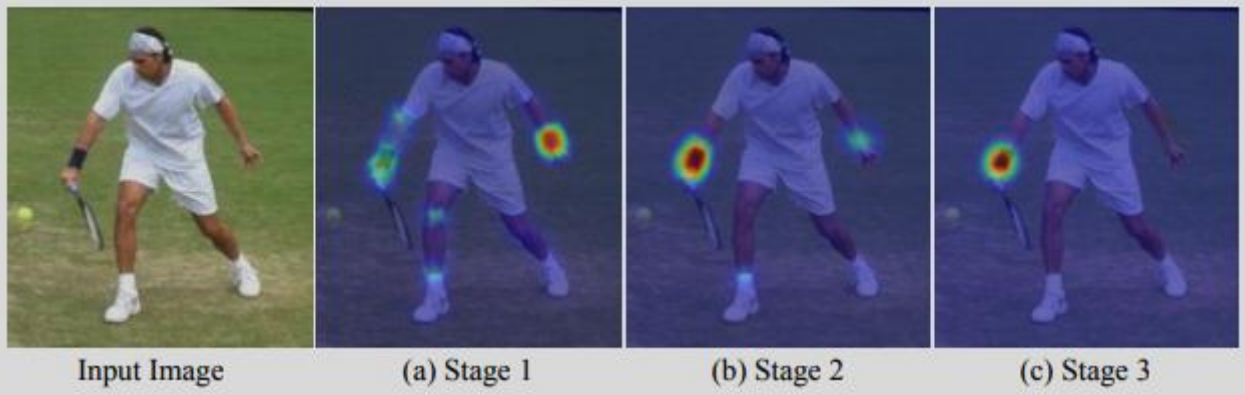
Head top: (x, y) → L2 loss

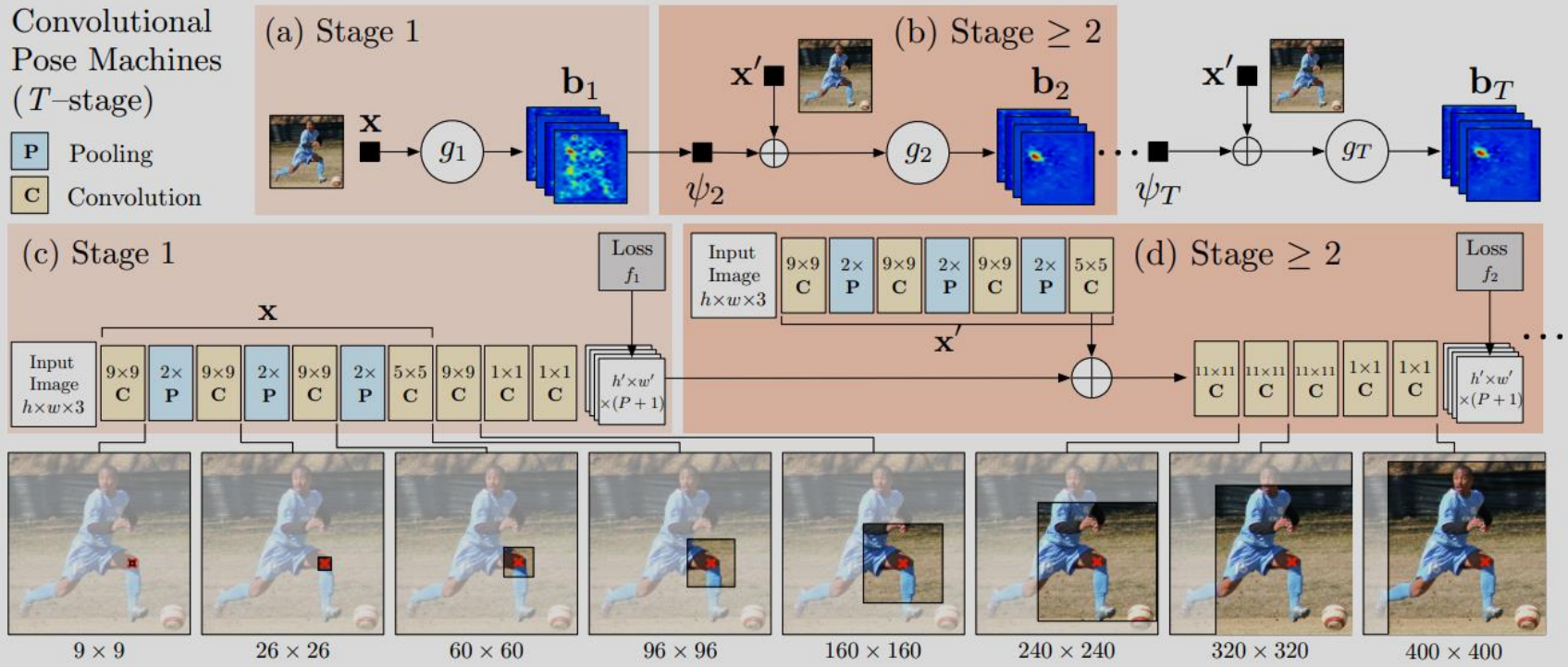
Correct left foot: (x', y')

Correct head top: (x', y')

+ → Loss

Pose estimation



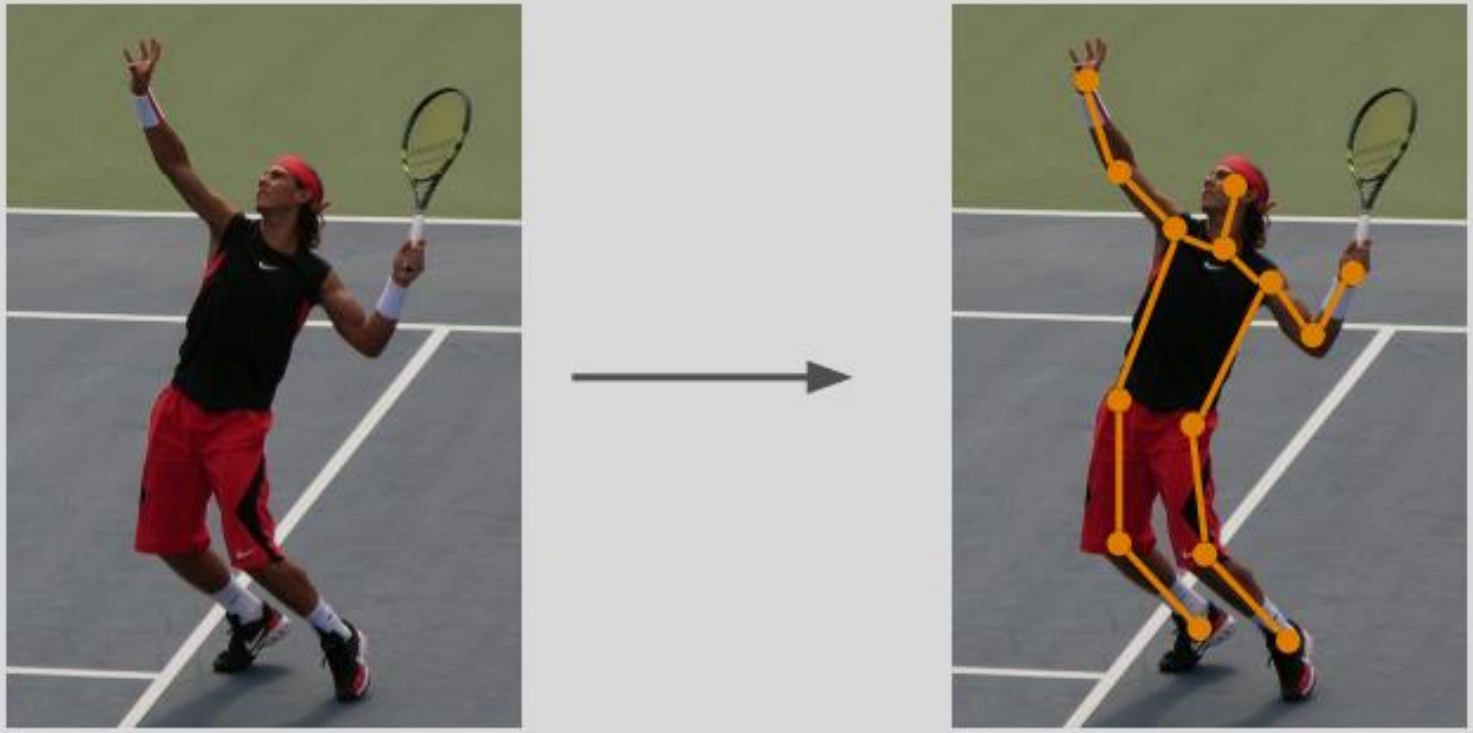


Pose estimation

```
class CPM2DPose(nn.Module):
    def __init__(self):
        super(CPM2DPose, self).__init__()

        self.relu = F.leaky_relu
        self.conv1_1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1, bias=True)
        self.conv1_2 = nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1, bias=True)
        self.conv2_1 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1, bias=True)
        self.conv2_2 = nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1, bias=True)
        self.conv3_1 = nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1, bias=True)
        self.conv3_2 = nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1, bias=True)
        self.conv3_3 = nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1, bias=True)
        self.conv3_4 = nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1, bias=True)
        self.conv4_1 = nn.Conv2d(256, 512, kernel_size=3, stride=1, padding=1, bias=True)
        self.conv4_2 = nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1, bias=True)
        self.conv4_3 = nn.Conv2d(512, 256, kernel_size=3, stride=1, padding=1, bias=True)
        self.conv4_4 = nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1, bias=True)
        self.conv4_5 = nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1, bias=True)
        self.conv4_6 = nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1, bias=True)
        self.conv4_7 = nn.Conv2d(256, 128, kernel_size=3, stride=1, padding=1, bias=True)
        self.conv5_1 = nn.Conv2d(128, 512, kernel_size=1, stride=1, padding=0, bias=True)
        self.conv5_2 = nn.Conv2d(512, 21, kernel_size=1, stride=1, padding=0, bias=True)
        self.conv6_1 = nn.Conv2d(149, 128, kernel_size=7, stride=1, padding=3, bias=True)
        self.conv6_2 = nn.Conv2d(128, 128, kernel_size=7, stride=1, padding=3, bias=True)
        self.conv6_3 = nn.Conv2d(128, 128, kernel_size=7, stride=1, padding=3, bias=True)
        self.conv6_4 = nn.Conv2d(128, 128, kernel_size=7, stride=1, padding=3, bias=True)
        self.conv6_5 = nn.Conv2d(128, 128, kernel_size=7, stride=1, padding=3, bias=True)
        self.conv6_6 = nn.Conv2d(128, 128, kernel_size=1, stride=1, padding=0, bias=True)
        self.conv6_7 = nn.Conv2d(128, 21, kernel_size=1, stride=1, padding=0, bias=True)
        self.conv7_1 = nn.Conv2d(149, 128, kernel_size=7, stride=1, padding=3, bias=True)
        self.conv7_2 = nn.Conv2d(128, 128, kernel_size=7, stride=1, padding=3, bias=True)
        self.conv7_3 = nn.Conv2d(128, 128, kernel_size=7, stride=1, padding=3, bias=True)
        self.conv7_4 = nn.Conv2d(128, 128, kernel_size=7, stride=1, padding=3, bias=True)
        self.conv7_5 = nn.Conv2d(128, 128, kernel_size=7, stride=1, padding=3, bias=True)
        self.conv7_6 = nn.Conv2d(128, 128, kernel_size=1, stride=1, padding=0, bias=True)
        self.conv7_7 = nn.Conv2d(128, 21, kernel_size=1, stride=1, padding=0, bias=True)
        self.maxpool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
```

```
def forward(self, x):
    x = self.relu(self.conv1_1(x))
    x = self.relu(self.conv1_2(x))
    x = self.maxpool(x)
    x = self.relu(self.conv2_1(x))
    x = self.relu(self.conv2_2(x))
    x = self.maxpool(x)
    x = self.relu(self.conv3_1(x))
    x = self.relu(self.conv3_2(x))
    x = self.relu(self.conv3_3(x))
    x = self.relu(self.conv3_4(x))
    x = self.maxpool(x)
    x = self.relu(self.conv4_1(x))
    x = self.relu(self.conv4_2(x))
    x = self.relu(self.conv4_3(x))
    x = self.relu(self.conv4_4(x))
    x = self.relu(self.conv4_5(x))
    x = self.relu(self.conv4_6(x))
    encoding = self.relu(self.conv4_7(x))
    x = self.relu(self.conv5_1(encoding))
    scoremap = self.conv5_2(x)

    x = torch.cat([scoremap, encoding],1)
    x = self.relu(self.conv6_1(x))
    x = self.relu(self.conv6_2(x))
    x = self.relu(self.conv6_3(x))
    x = self.relu(self.conv6_4(x))
    x = self.relu(self.conv6_5(x))
    x = self.relu(self.conv6_6(x))
    scoremap = self.conv6_7(x)

    x = torch.cat([scoremap, encoding], 1)
    x = self.relu(self.conv7_1(x))
    x = self.relu(self.conv7_2(x))
    x = self.relu(self.conv7_3(x))
    x = self.relu(self.conv7_4(x))
    x = self.relu(self.conv7_5(x))
    x = self.relu(self.conv7_6(x))
    x = self.conv7_7(x)
    return x
```

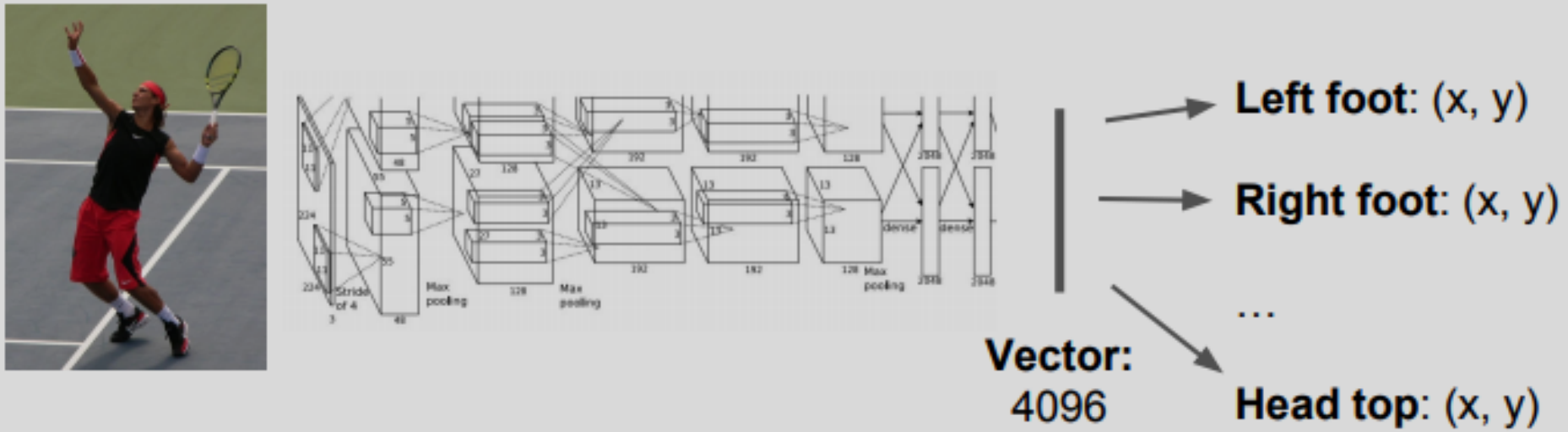
Pose estimation



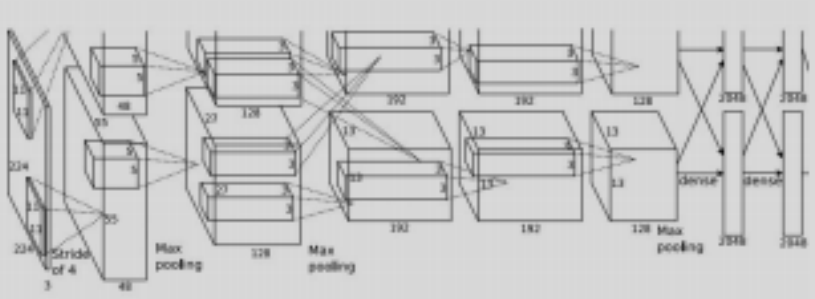
Represent pose as a set of 14 joint positions:

Left / right foot
Left / right knee
Left / right hip
Left / right shoulder
Left / right elbow
Left / right hand
Neck
Head top

Pose estimation



Pose estimation



Vector:
4096

Left foot: (x, y) → L2 loss

Right foot: (x, y) → L2 loss

…

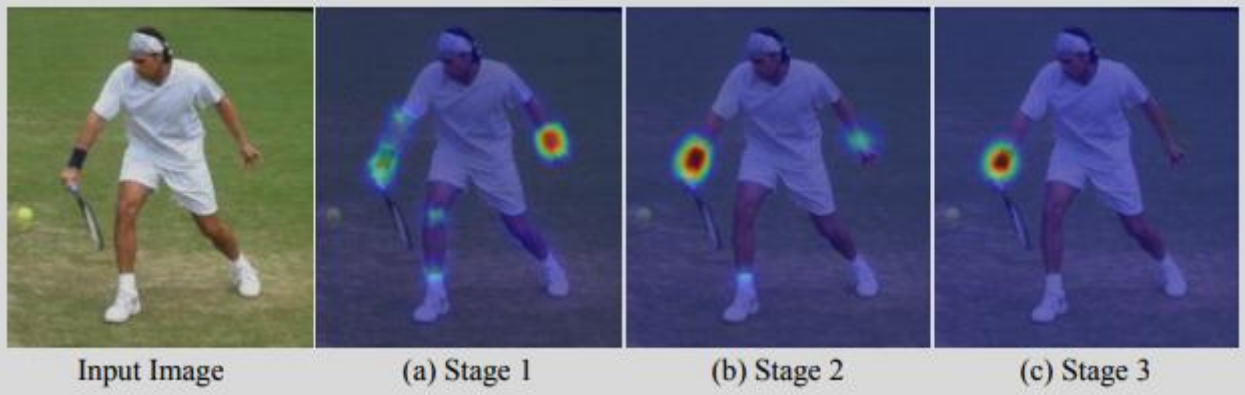
Head top: (x, y) → L2 loss

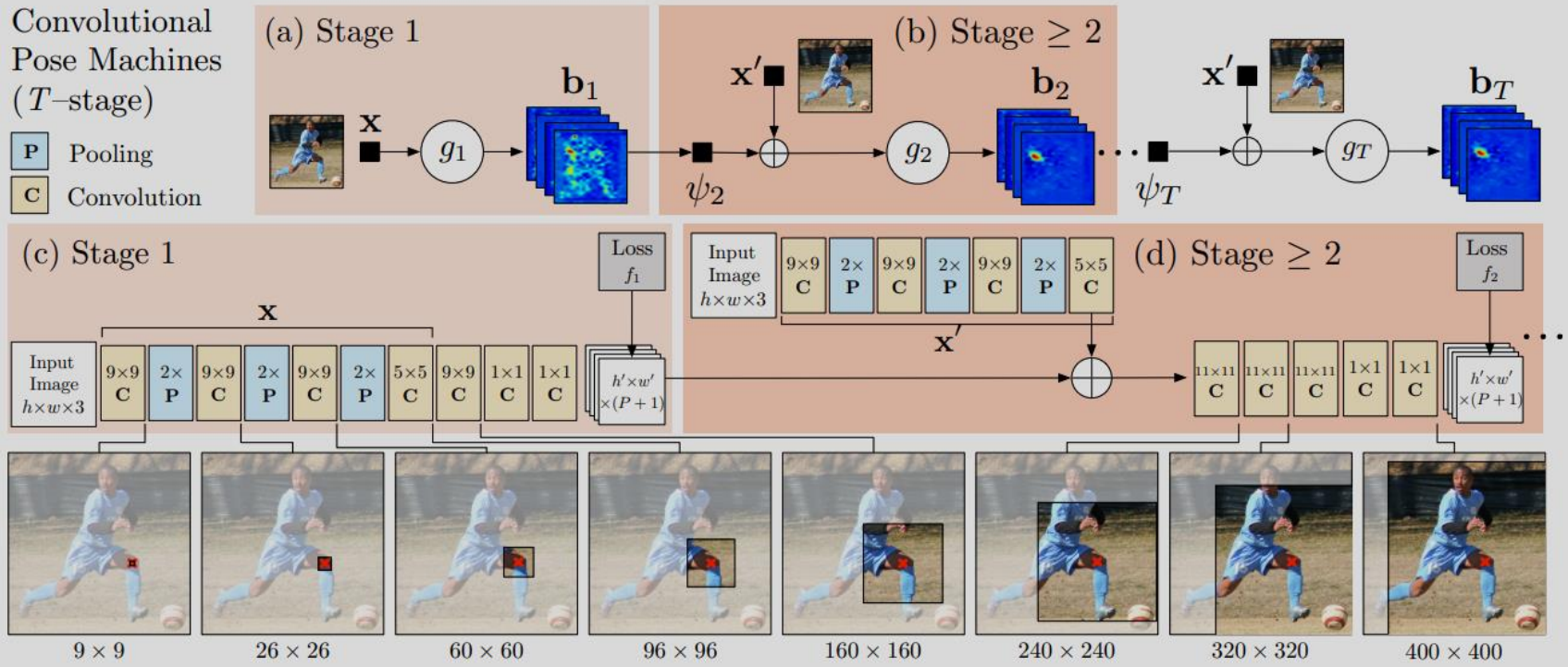
Correct left foot: (x', y')

Correct head top: (x', y')

+ → Loss

Pose estimation





Pose estimation

```
class CPM2DPose(nn.Module):
    def __init__(self):
        super(CPM2DPose, self).__init__()

        self.relu = F.leaky_relu
        self.conv1_1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1, bias=True)
        self.conv1_2 = nn.Conv2d(64, 64, kernel_size=3, stride=1, padding=1, bias=True)
        self.conv2_1 = nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1, bias=True)
        self.conv2_2 = nn.Conv2d(128, 128, kernel_size=3, stride=1, padding=1, bias=True)
        self.conv3_1 = nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1, bias=True)
        self.conv3_2 = nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1, bias=True)
        self.conv3_3 = nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1, bias=True)
        self.conv3_4 = nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1, bias=True)
        self.conv4_1 = nn.Conv2d(256, 512, kernel_size=3, stride=1, padding=1, bias=True)
        self.conv4_2 = nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1, bias=True)
        self.conv4_3 = nn.Conv2d(512, 256, kernel_size=3, stride=1, padding=1, bias=True)
        self.conv4_4 = nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1, bias=True)
        self.conv4_5 = nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1, bias=True)
        self.conv4_6 = nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1, bias=True)
        self.conv4_7 = nn.Conv2d(256, 128, kernel_size=3, stride=1, padding=1, bias=True)
        self.conv5_1 = nn.Conv2d(128, 512, kernel_size=1, stride=1, padding=0, bias=True)
        self.conv5_2 = nn.Conv2d(512, 21, kernel_size=1, stride=1, padding=0, bias=True)
        self.conv6_1 = nn.Conv2d(149, 128, kernel_size=7, stride=1, padding=3, bias=True)
        self.conv6_2 = nn.Conv2d(128, 128, kernel_size=7, stride=1, padding=3, bias=True)
        self.conv6_3 = nn.Conv2d(128, 128, kernel_size=7, stride=1, padding=3, bias=True)
        self.conv6_4 = nn.Conv2d(128, 128, kernel_size=7, stride=1, padding=3, bias=True)
        self.conv6_5 = nn.Conv2d(128, 128, kernel_size=7, stride=1, padding=3, bias=True)
        self.conv6_6 = nn.Conv2d(128, 128, kernel_size=1, stride=1, padding=0, bias=True)
        self.conv6_7 = nn.Conv2d(128, 21, kernel_size=1, stride=1, padding=0, bias=True)
        self.conv7_1 = nn.Conv2d(149, 128, kernel_size=7, stride=1, padding=3, bias=True)
        self.conv7_2 = nn.Conv2d(128, 128, kernel_size=7, stride=1, padding=3, bias=True)
        self.conv7_3 = nn.Conv2d(128, 128, kernel_size=7, stride=1, padding=3, bias=True)
        self.conv7_4 = nn.Conv2d(128, 128, kernel_size=7, stride=1, padding=3, bias=True)
        self.conv7_5 = nn.Conv2d(128, 128, kernel_size=7, stride=1, padding=3, bias=True)
        self.conv7_6 = nn.Conv2d(128, 128, kernel_size=1, stride=1, padding=0, bias=True)
        self.conv7_7 = nn.Conv2d(128, 21, kernel_size=1, stride=1, padding=0, bias=True)
        self.maxpool = nn.MaxPool2d(kernel_size=2, stride=2, padding=0)
```

```
def forward(self, x):
    x = self.relu(self.conv1_1(x))
    x = self.relu(self.conv1_2(x))
    x = self.maxpool(x)
    x = self.relu(self.conv2_1(x))
    x = self.relu(self.conv2_2(x))
    x = self.maxpool(x)
    x = self.relu(self.conv3_1(x))
    x = self.relu(self.conv3_2(x))
    x = self.relu(self.conv3_3(x))
    x = self.relu(self.conv3_4(x))
    x = self.maxpool(x)
    x = self.relu(self.conv4_1(x))
    x = self.relu(self.conv4_2(x))
    x = self.relu(self.conv4_3(x))
    x = self.relu(self.conv4_4(x))
    x = self.relu(self.conv4_5(x))
    x = self.relu(self.conv4_6(x))
    encoding = self.relu(self.conv4_7(x))
    x = self.relu(self.conv5_1(encoding))
    scoremap = self.conv5_2(x)

    x = torch.cat([scoremap, encoding],1)
    x = self.relu(self.conv6_1(x))
    x = self.relu(self.conv6_2(x))
    x = self.relu(self.conv6_3(x))
    x = self.relu(self.conv6_4(x))
    x = self.relu(self.conv6_5(x))
    x = self.relu(self.conv6_6(x))
    scoremap = self.conv6_7(x)

    x = torch.cat([scoremap, encoding], 1)
    x = self.relu(self.conv7_1(x))
    x = self.relu(self.conv7_2(x))
    x = self.relu(self.conv7_3(x))
    x = self.relu(self.conv7_4(x))
    x = self.relu(self.conv7_5(x))
    x = self.relu(self.conv7_6(x))
    x = self.conv7_7(x)
    return x
```

Object detection

Multi-scale windows

Sliding Window

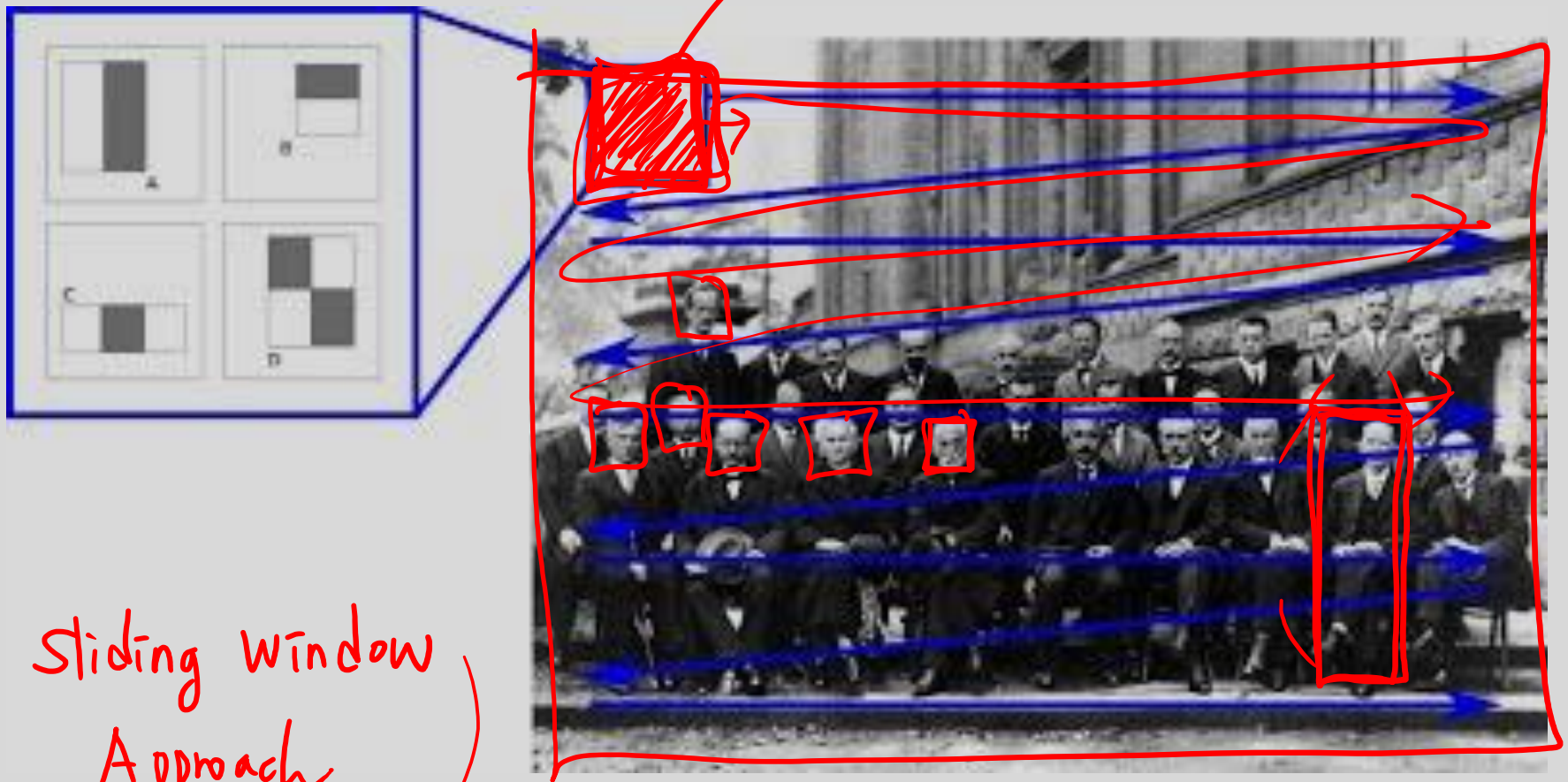
640x480x # of B.B

480.

few seconds

Considering the multi-scales, sliding window is too slow.

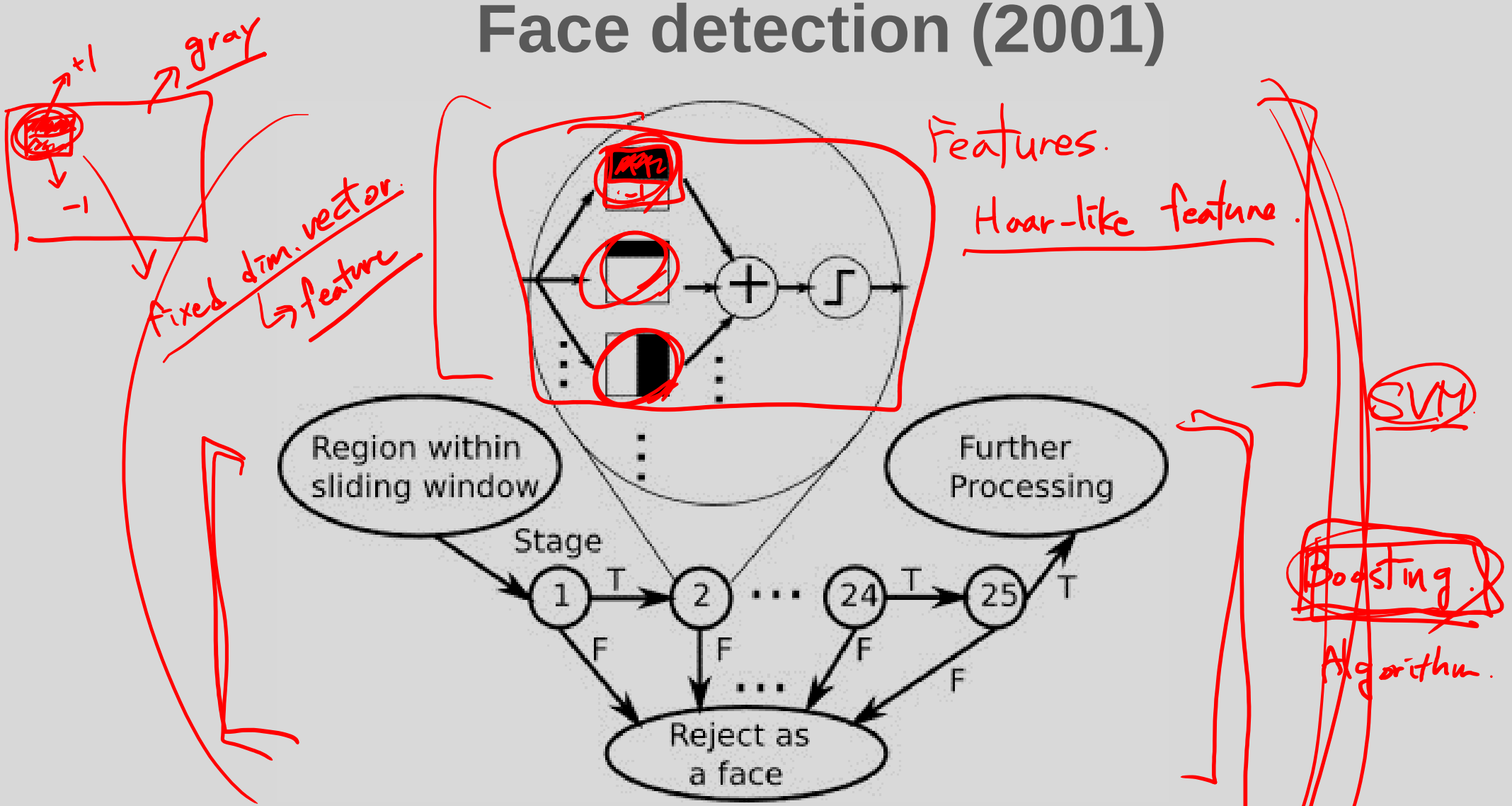
Face detection (2001)



Sliding Window Approach

Viola & Jones, CVPR'01

Face detection (2001)



Viola & Jones, CVPR'01

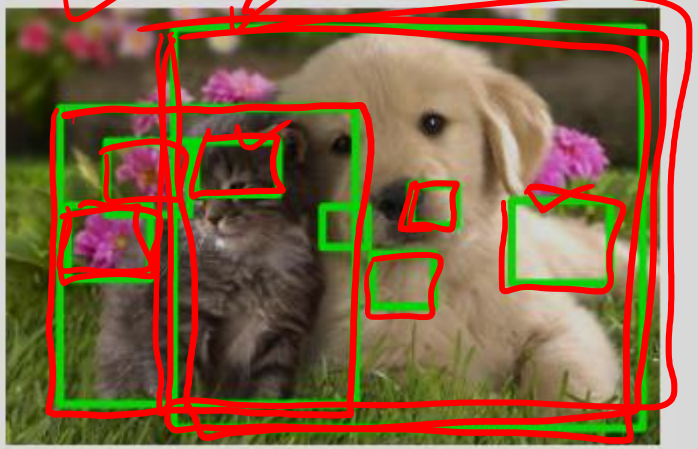
Object proposal

Find image regions that are likely to contain objects.

E.g. Selective search. (1000 regions in a few seconds on CPU).

B.B.
2~5

candidate B.B.



Assumption

If color / texture are similar,
there's high prob. that those pixels are belonging to same object.

Object proposal

Input Image

Output Image

Oversegmented Image

Considering Color, Texture, Size, Shape similarities.

Object proposal

Algorithm 1: Hierarchical Grouping Algorithm

Input: (colour) image
Output: Set of object location hypotheses L

Obtain initial regions $R = \{r_1, \cdots, r_n\}$ using [13]
Initialise similarity set $S = \emptyset$
foreach *Neighbouring region pair* (r_i, r_j) **do**
  Calculate similarity $s(r_i, r_j)$
  $S = S \cup s(r_i, r_j)$

while $S \neq \emptyset$ **do**
  Get highest similarity $s(r_i, r_j) = \max(S)$
  Merge corresponding regions $r_t = r_i \cup r_j$
  Remove similarities regarding $r_i : S = S \setminus s(r_i, r_*)$
  Remove similarities regarding $r_j : S = S \setminus s(r_*, r_j)$
  Calculate similarity set S_t between r_t and its neighbours
  $S = S \cup S_t$
  $R = R \cup r_t$

Extract object location boxes L from all regions in R

Object proposal

```
import cv2
from google.colab.patches import cv2_imshow
import random

image = cv2.imread("/content/unist.jpg")

ss = cv2.ximgproc.segmentation.createSelectiveSearchSegmentation()
ss.setBaseImage(image)
ss.switchToSelectiveSearchFast()
rects = ss.process()

for i in range(0, len(rects), 100):
  output = image.copy()
  for (x, y, w, h) in rects[i:i + 100]:
    color = [random.randint(0, 255) for j in range(0, 3)]
    cv2.rectangle(output, (x, y), (x + w, y + h), color, 2)
cv2_imshow(output)
```

1000 ~ 2000

2 ~ 5

R-CNN (CVPR'13)

Regional

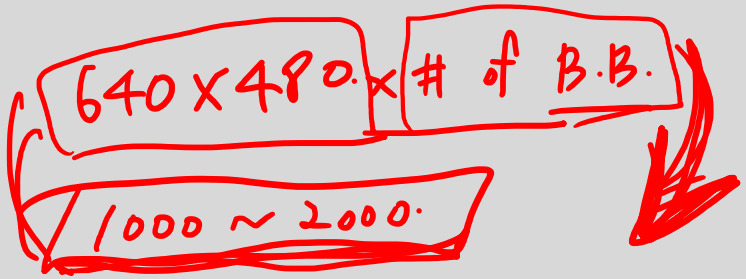
640x480x # of B.B types.

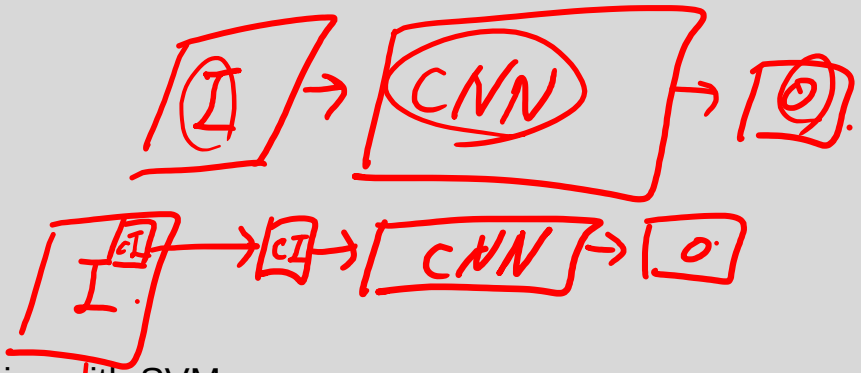
Input image

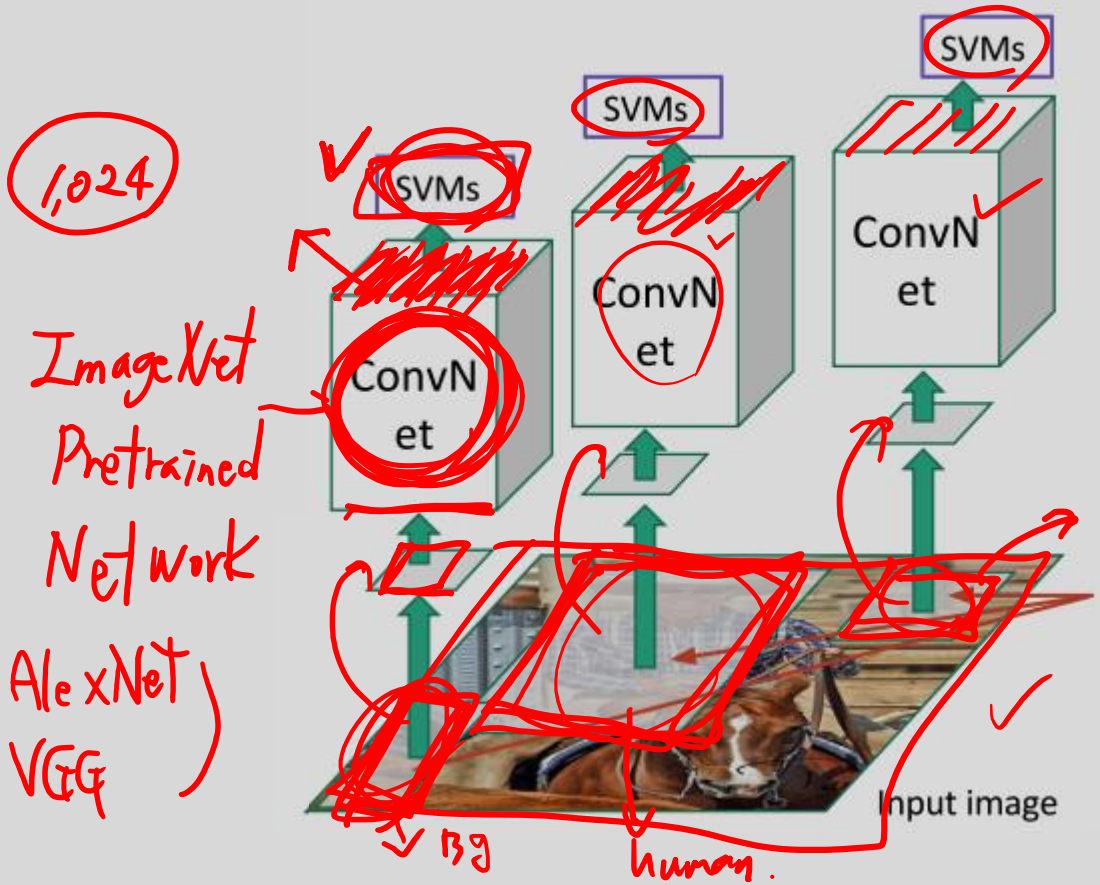
R-CNN (CVPR'13)

Regions of interest
From selective search (~2000).

R-CNN (CVPR'13)







Classify each region with SVMs.

whole Image

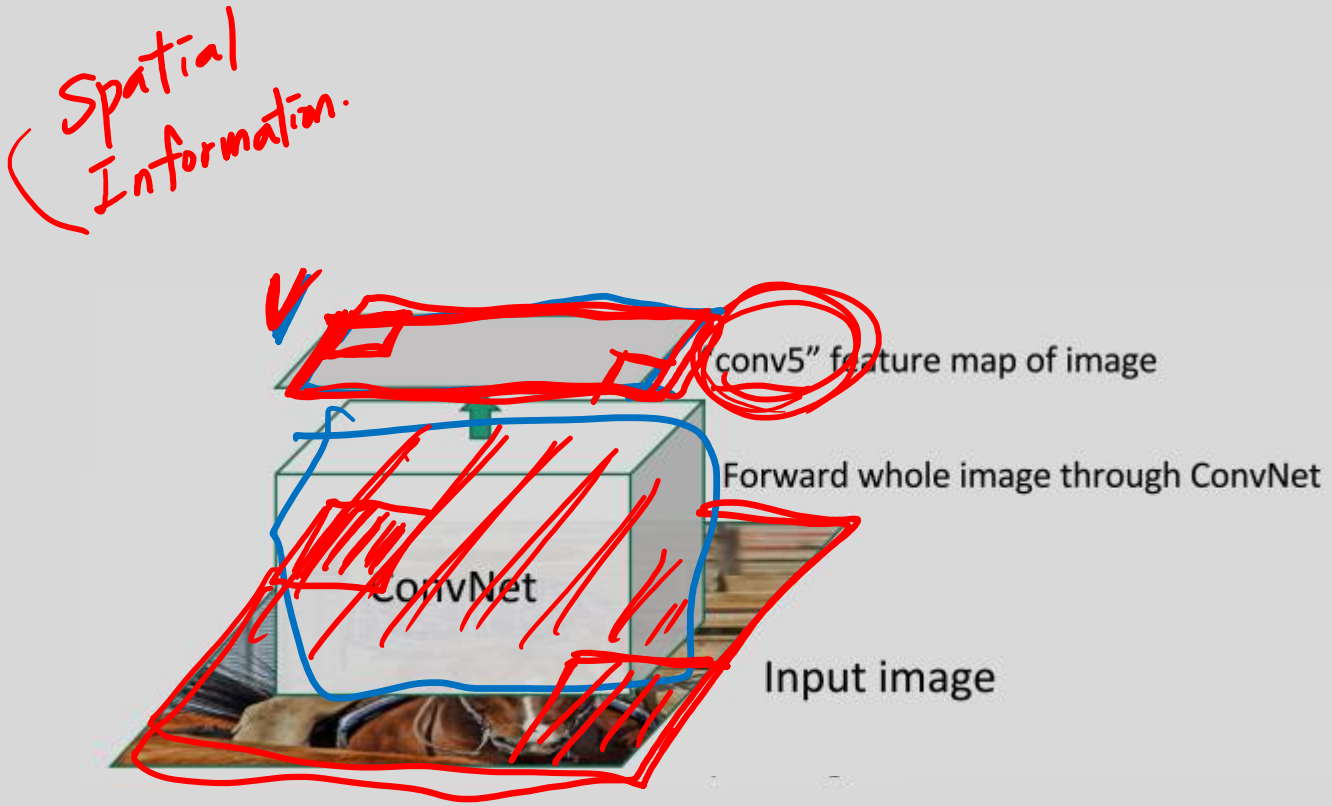
Forward with ImageNet-trained CNNs.

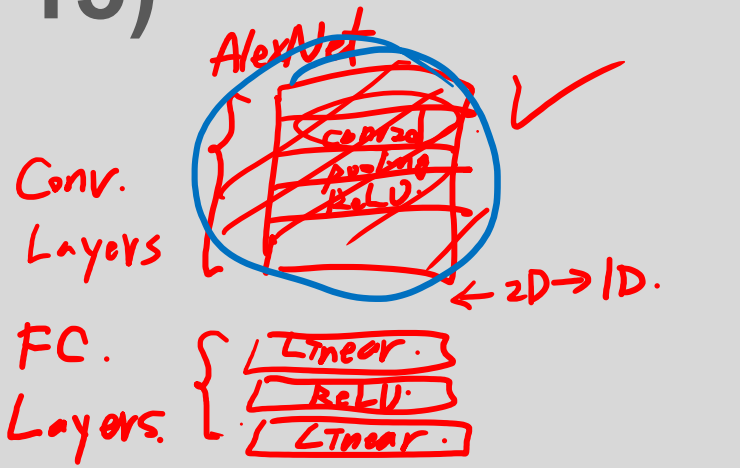
Regions of interest
From selective search (~2000).

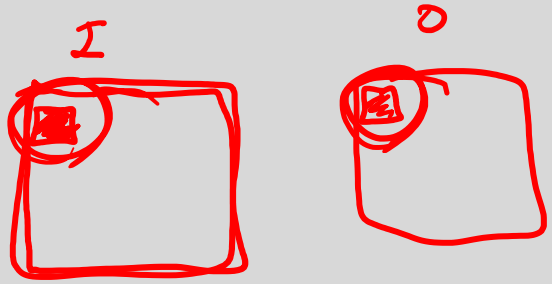
1000 ~2000

Limitations:
1) Not end-to-end trainable.
2) Still slow.

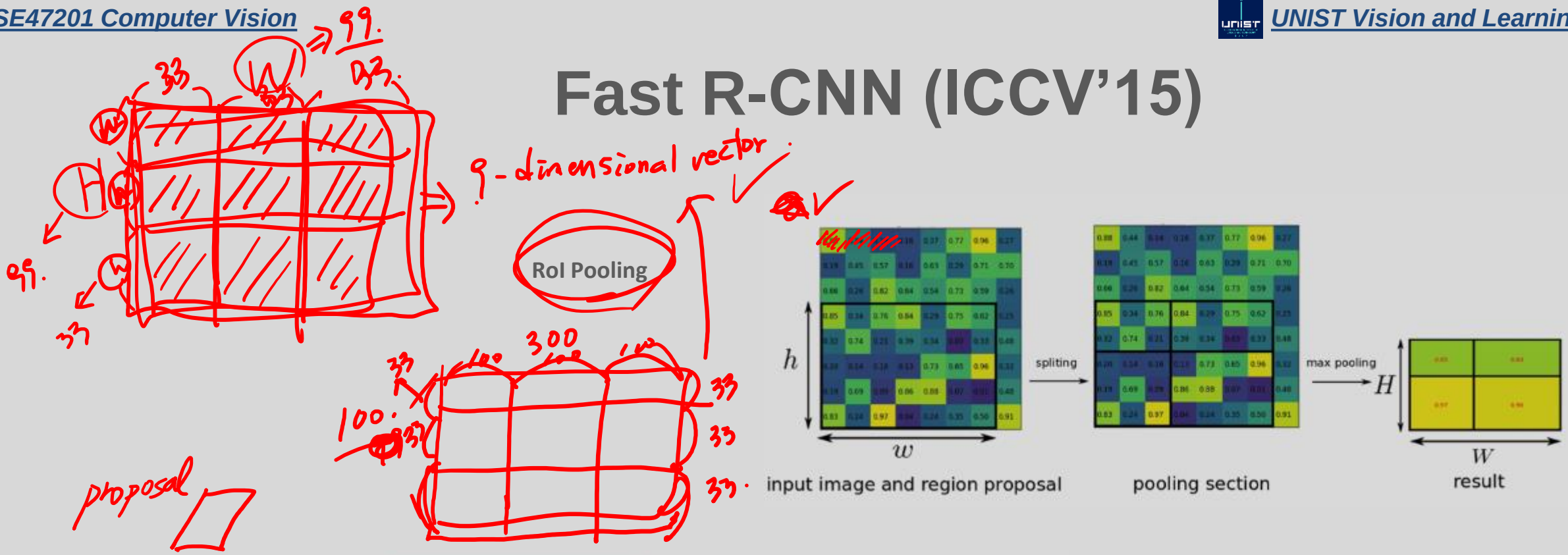
Fast R-CNN (ICCV'15)

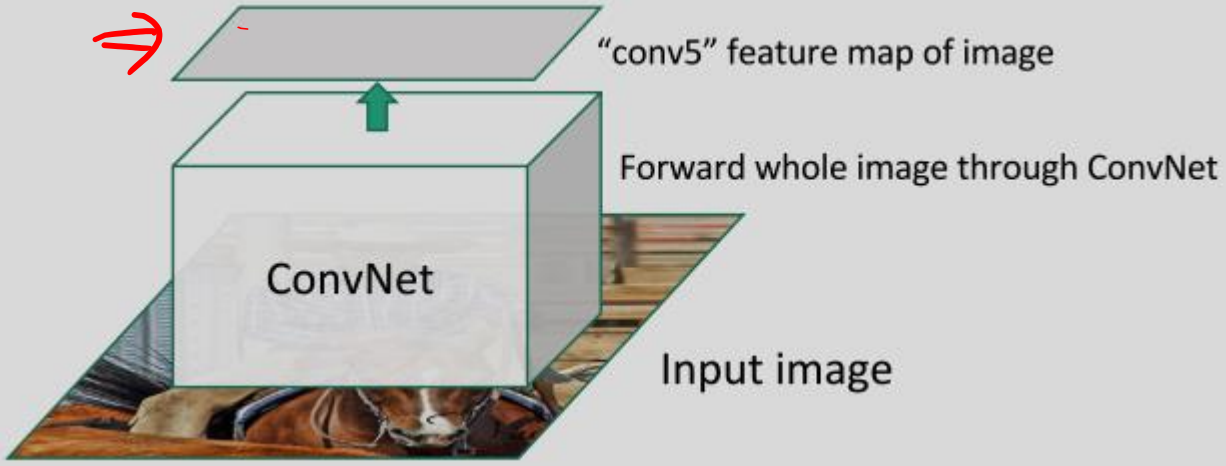




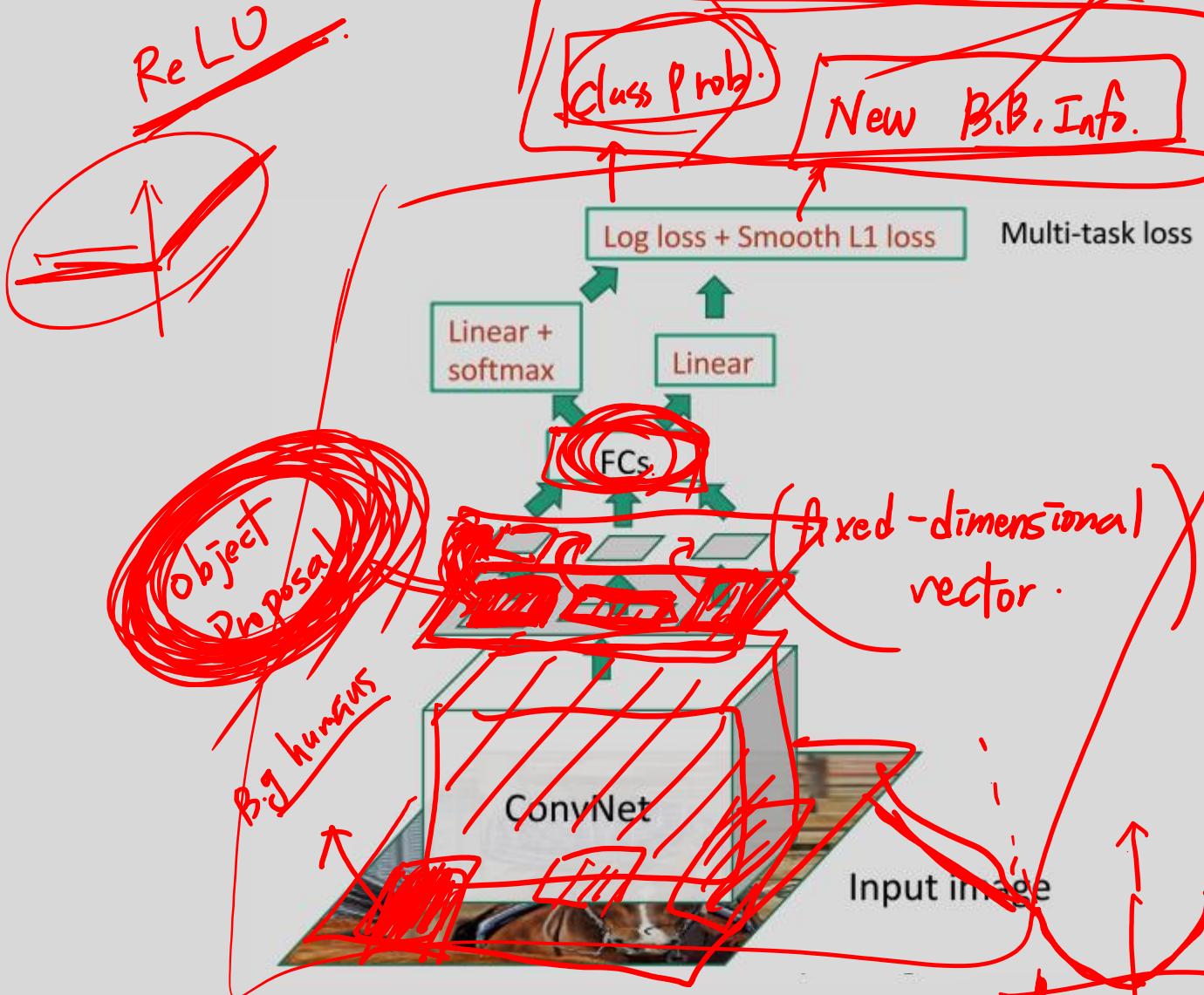


Fast R-CNN (ICCV'15)





Fast R-CNN (ICCV'15)



$$L(p, u, t^u, v) = L_{\text{cls}}(p, u) + \lambda[u \geq 1] L_{\text{loc}}(t^u, v)$$

$$p = (p_0, \ldots, p_K)$$

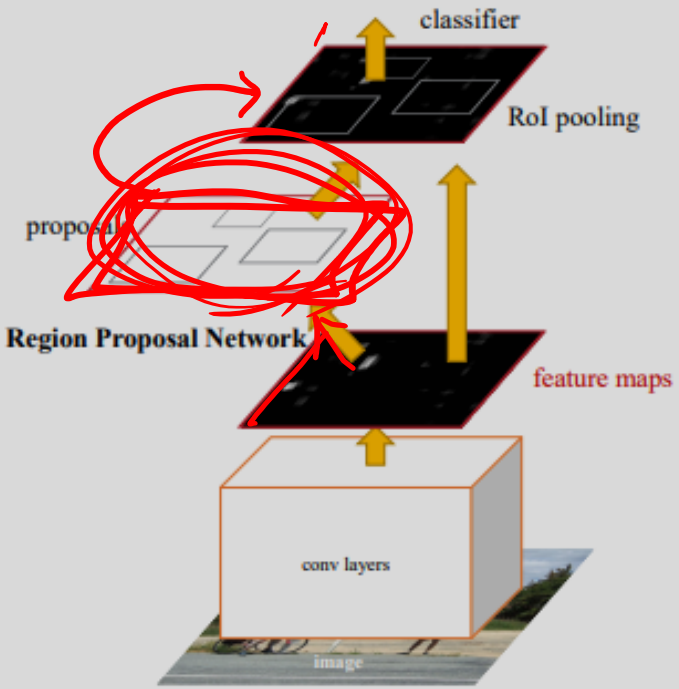
$$t^u = (t^u_{\text{x}}, t^u_{\text{y}}, t^u_{\text{w}}, t^u_{\text{h}})$$

$$L_{\text{cls}}(p, u) = -\log p_u$$

$$L_{\text{loc}}(t^u, v) = \sum_{i \in \{\text{x,y,w,h}\}} \text{smooth}_{L_1}(t^u_i - v_i)$$

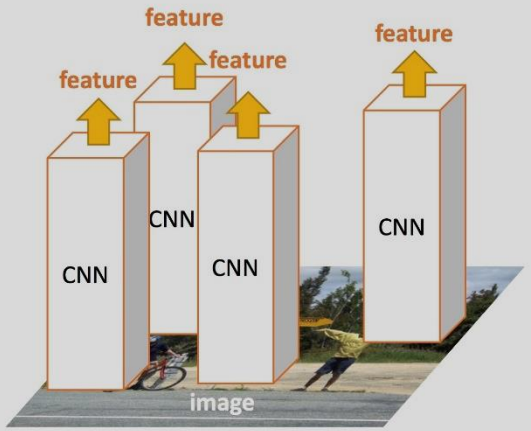
$$\text{smooth}_{L_1}(x) = \begin{cases} 0.5x^2 & \text{if } |x| < 1 \\ |x| - 0.5 & \text{otherwise,} \end{cases}$$

Faster R-CNN (NIPS'15)



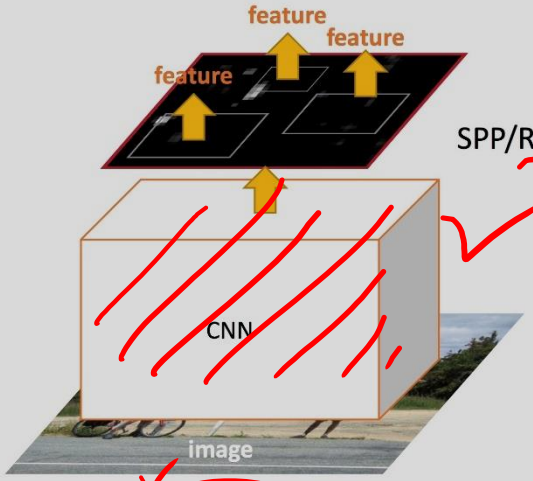
Solve the bottleneck in the region proposal of the Fast-RCNN

Comparisons



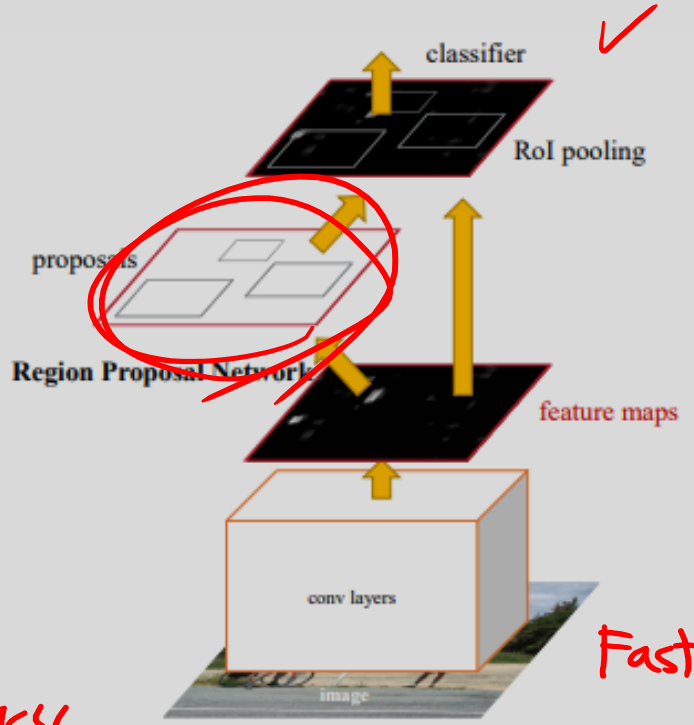
R-CNN

- 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



FasterRCNN.

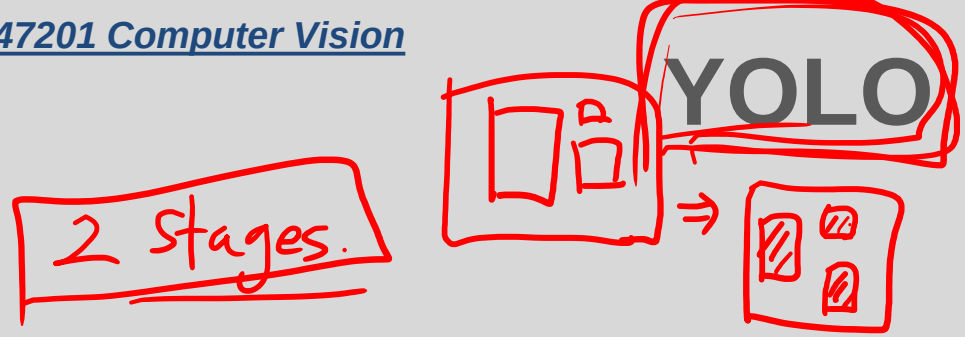
accuracy.

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**

x25
x10

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

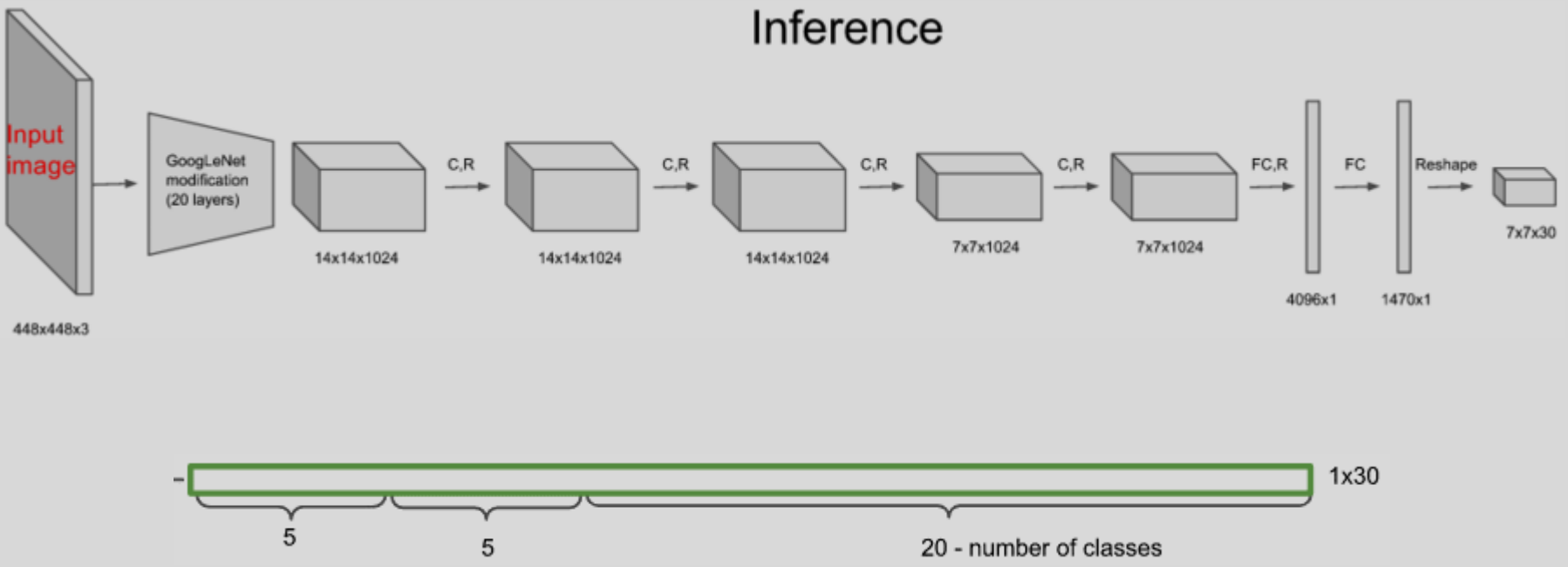
YOLO (You only look once, CVPR'16)



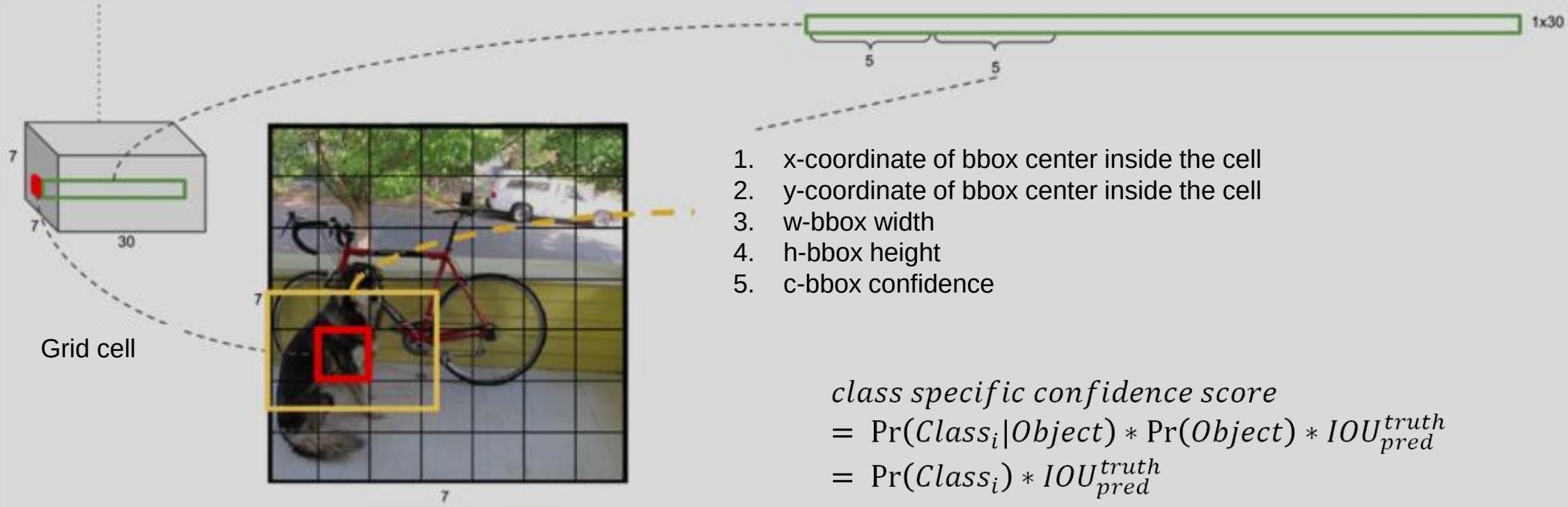
- 1 stage algorithm

YOLO (You only look once, CVPR'16)

- Two candidates for each grid, 20 classes:



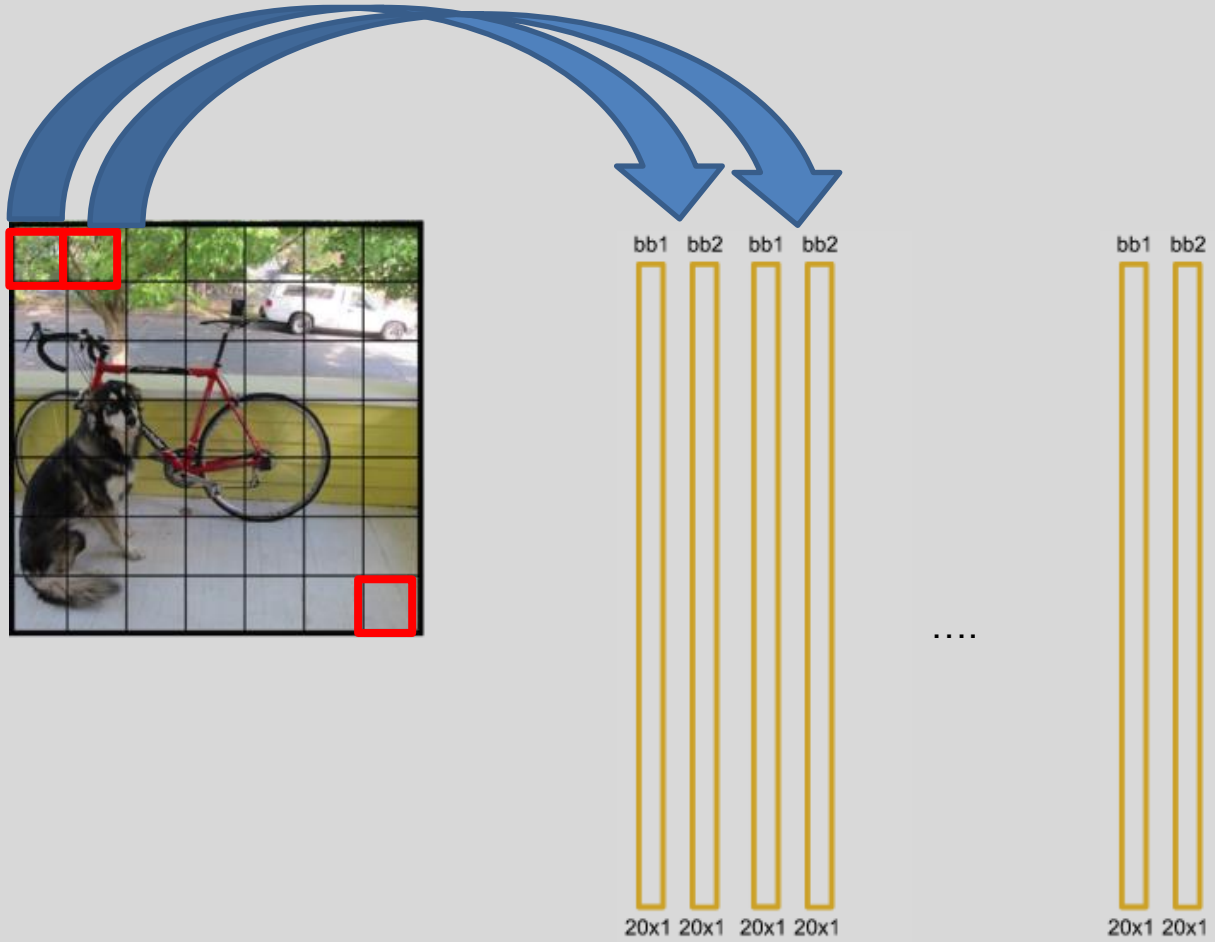
YOLO (You only look once, CVPR'16)



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

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

YOLO (You only look once, CVPR'16)



7x7x2=98 boxes

Loss

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

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

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

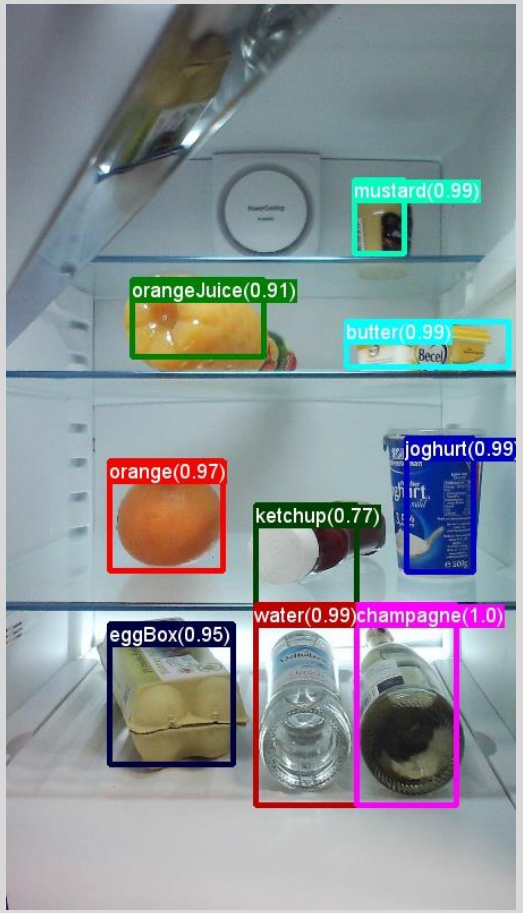
$$+ \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i \right)^2$$

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

YOLO (You only look once, CVPR'16)

- 1 stage algorithm

Non-maximal suppression



Non-maximal suppression

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

YOLO (You only look once, CVPR'16)

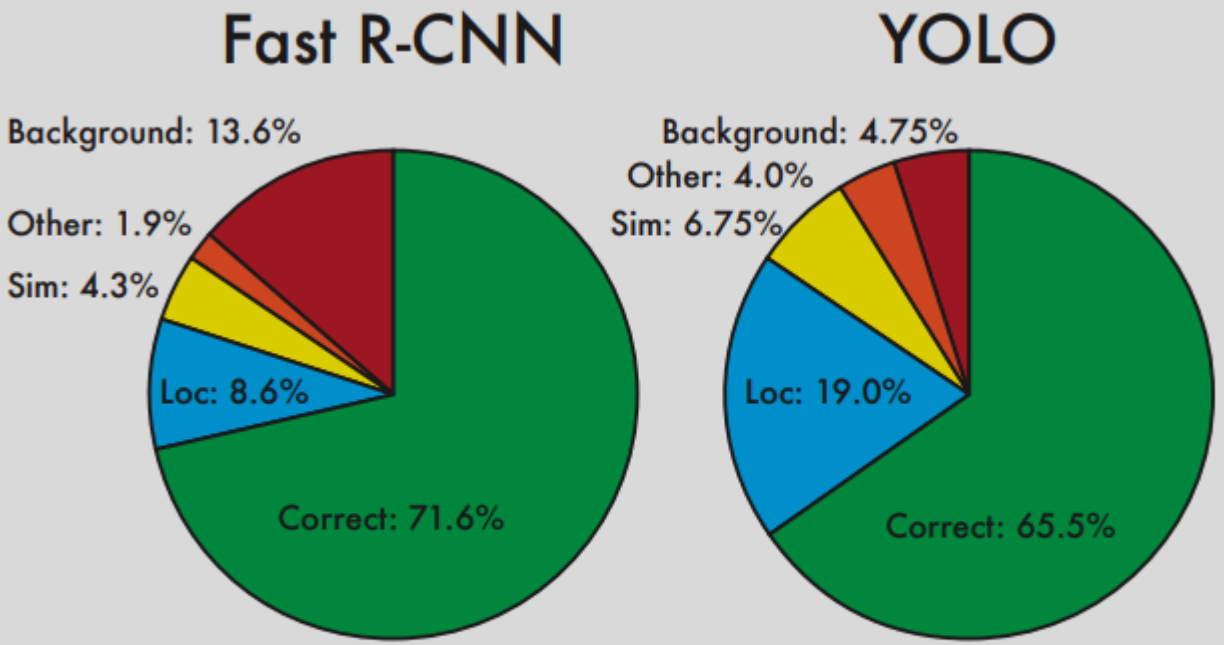
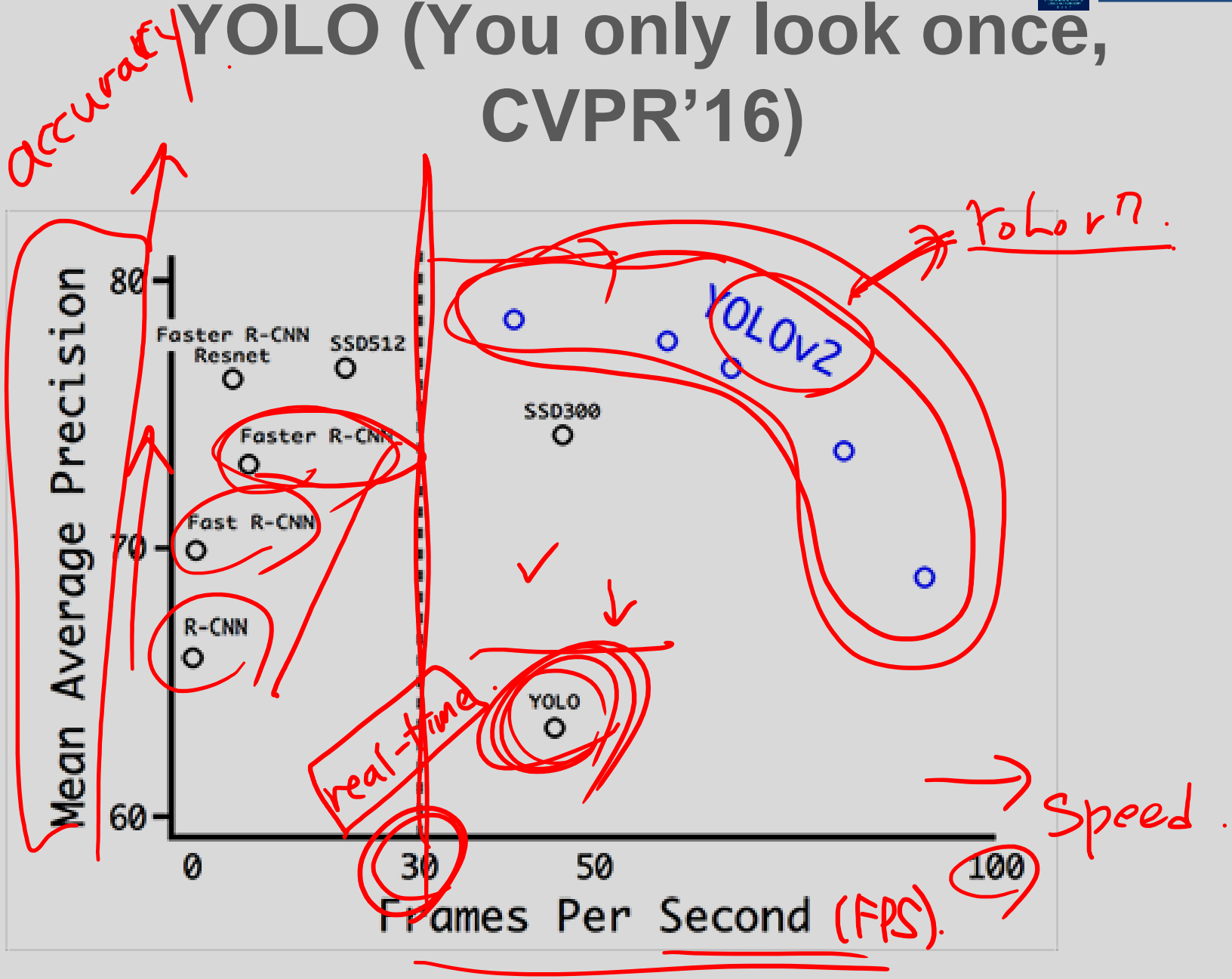


Figure 4: Error Analysis: Fast R-CNN vs. YOLO These charts show the percentage of localization and background errors in the top N detections for various categories (N = # objects in that category).

YOLO (You only look once, CVPR'16)

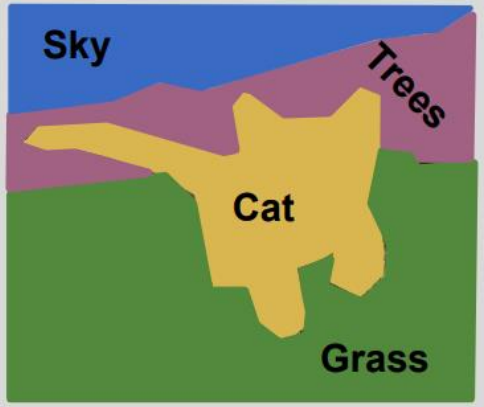


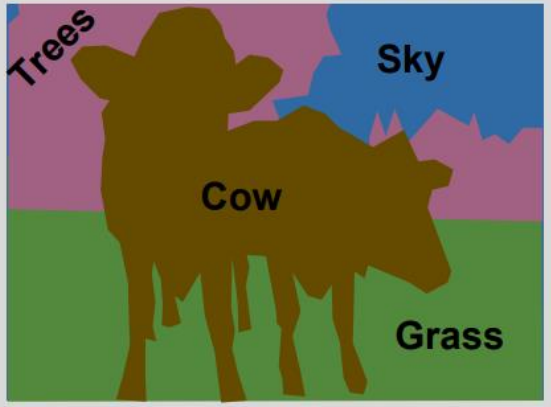
Semantic segmentation

Semantic Segmentation

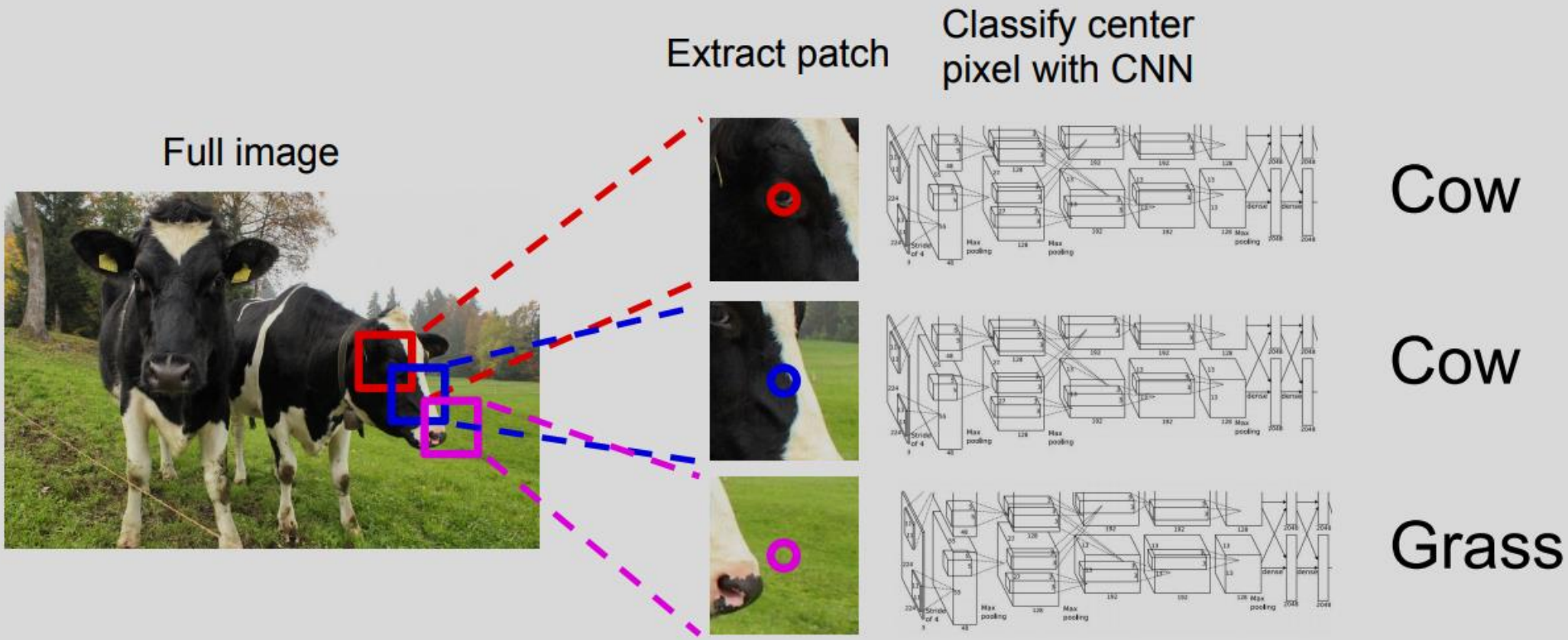
Label each pixel in the image with a category label

Don’t differentiate instances, only care about pixels



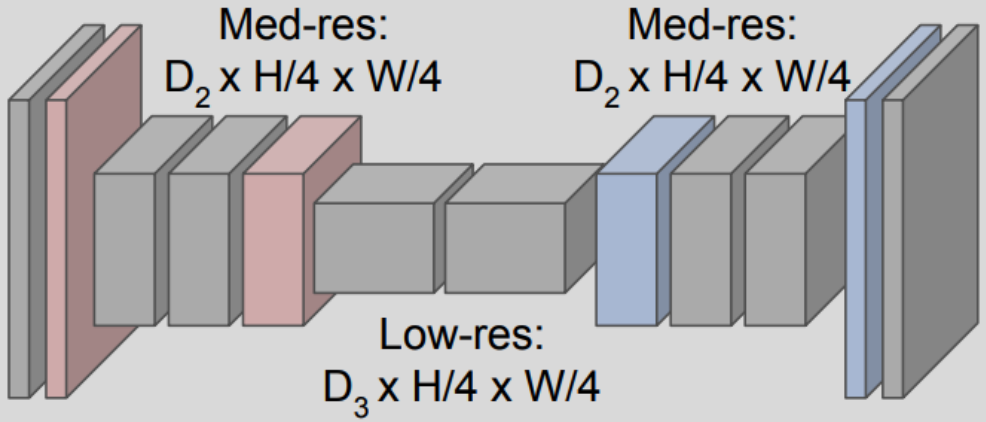


Semantic segmentation



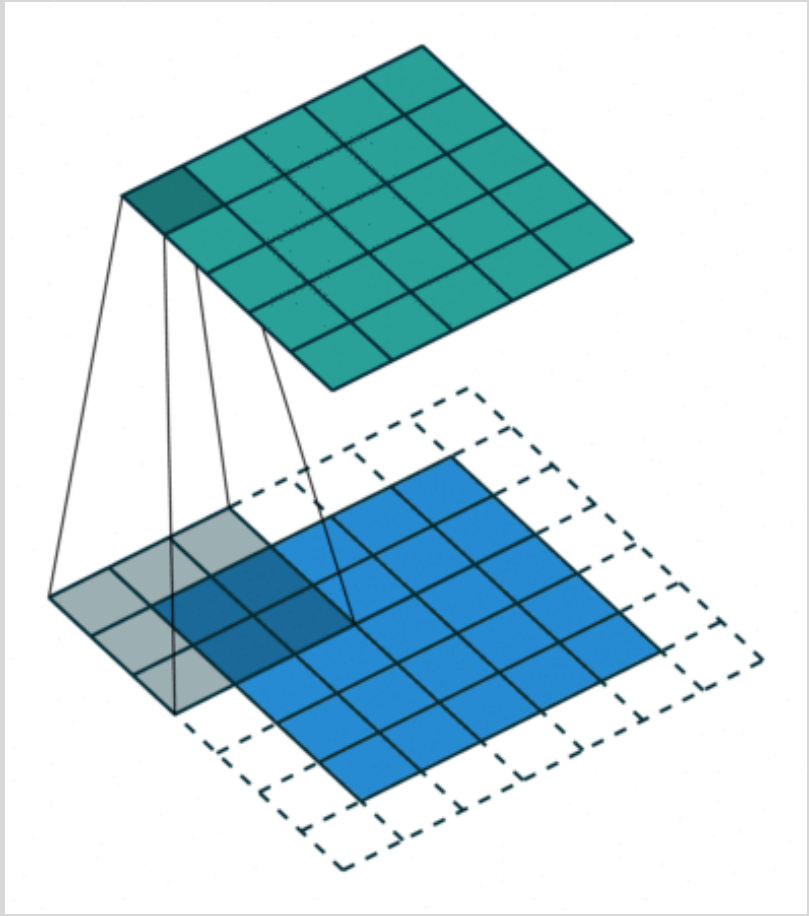
Semantic segmentation

Input:
3 x H x W

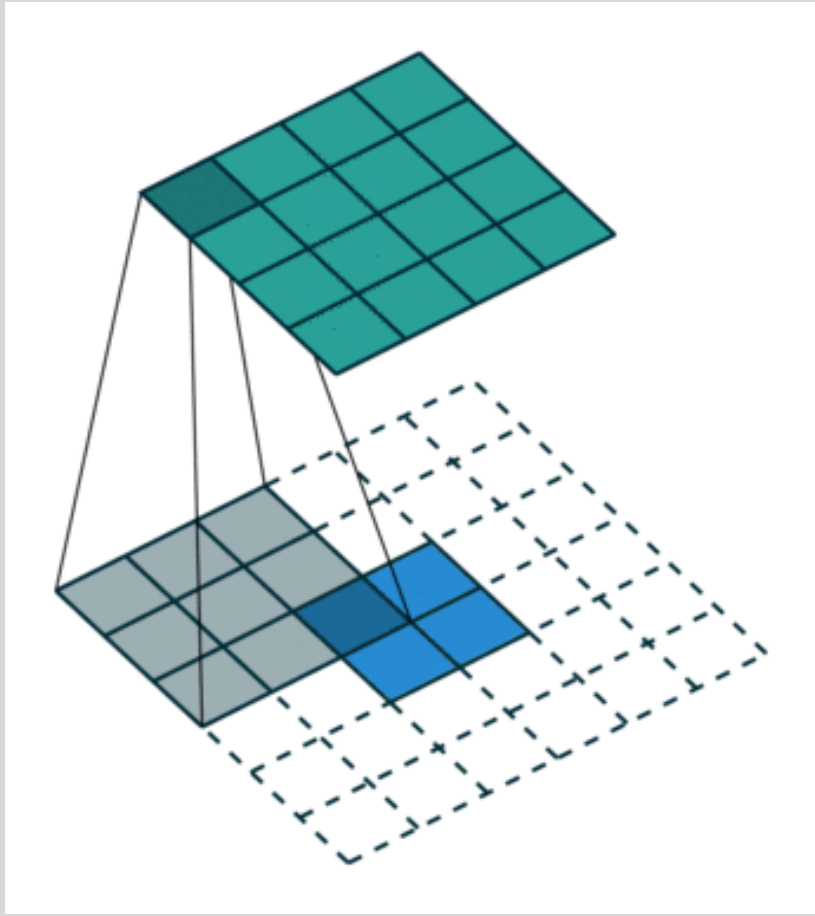


Predictions:
H x W

Transposed Convolution



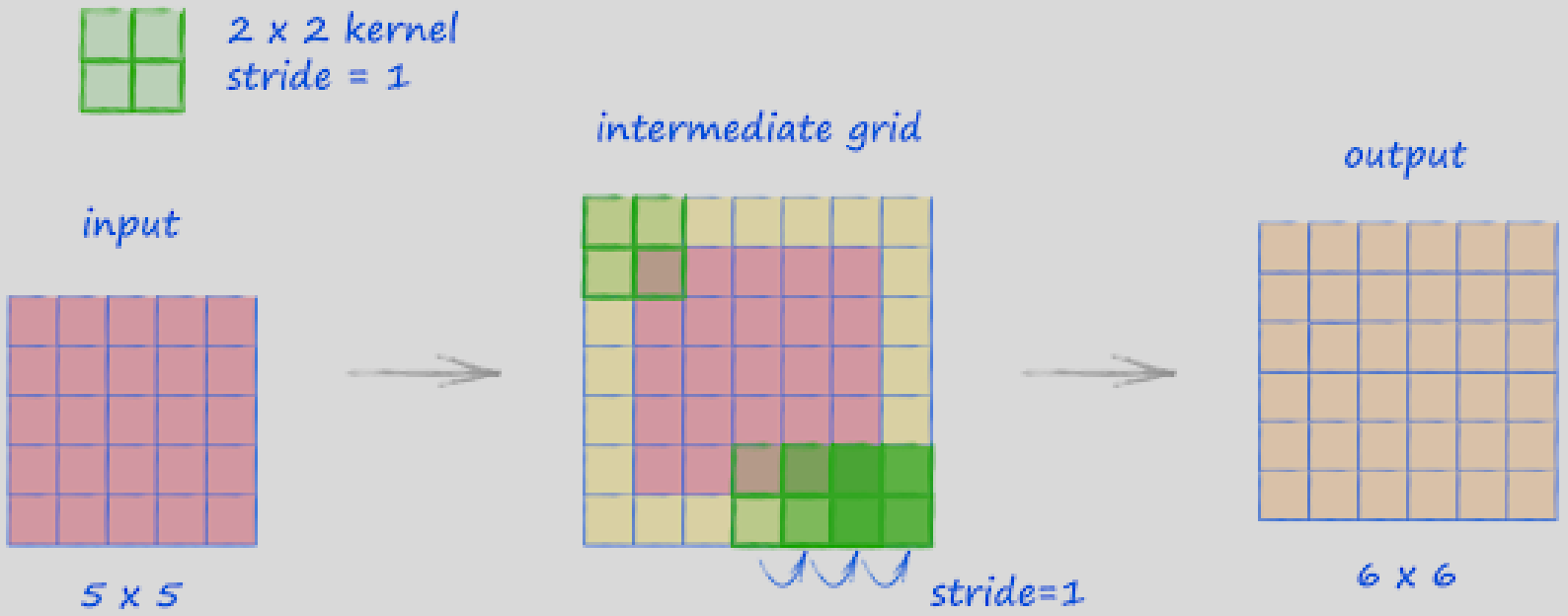
Convolution operation



Transposed convolution operation

Transposed Convolution

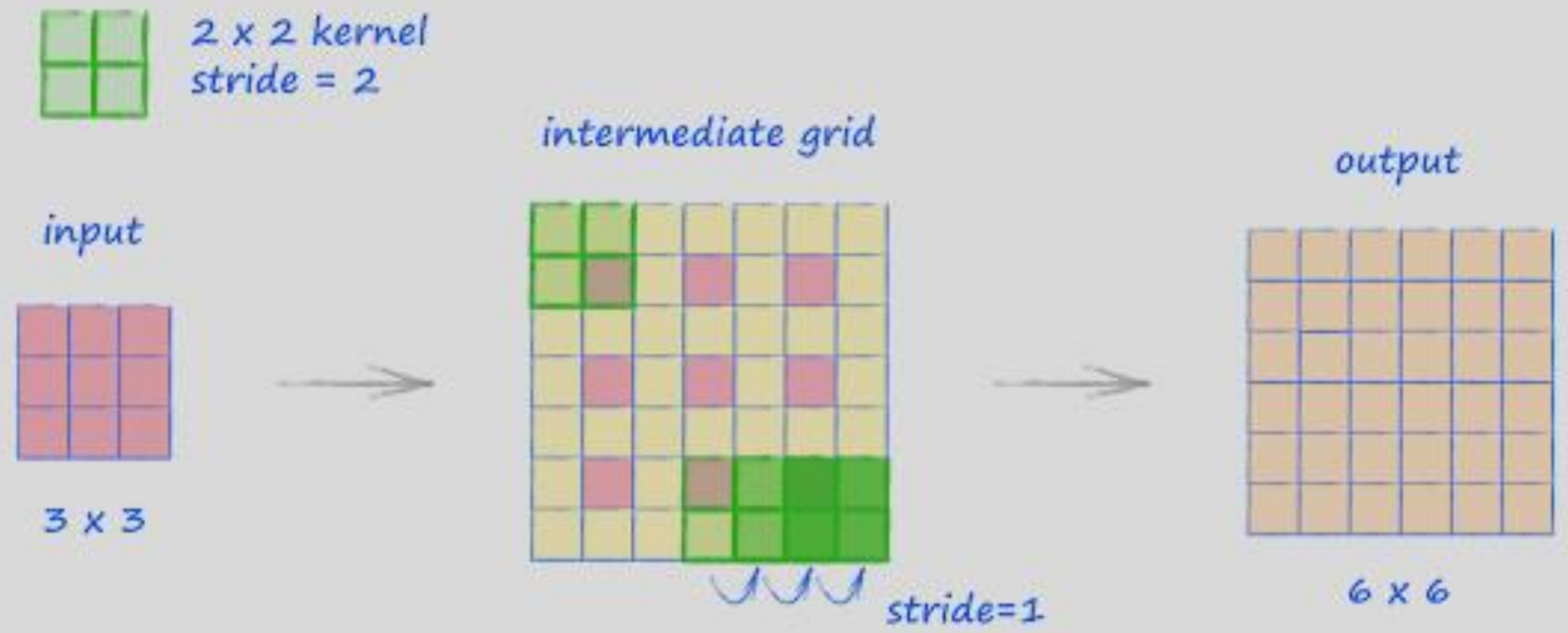
Transposed Convolution with 0 padding, stride 1, 2x2 kernel: Output_size = (input_size-1)*stride – 2*padding + kernel_size + output_padding




```
nn.ConvTranspose2d(in_channels, out_channels, kernel_size=2, stride=1)
```

Transposed Convolution

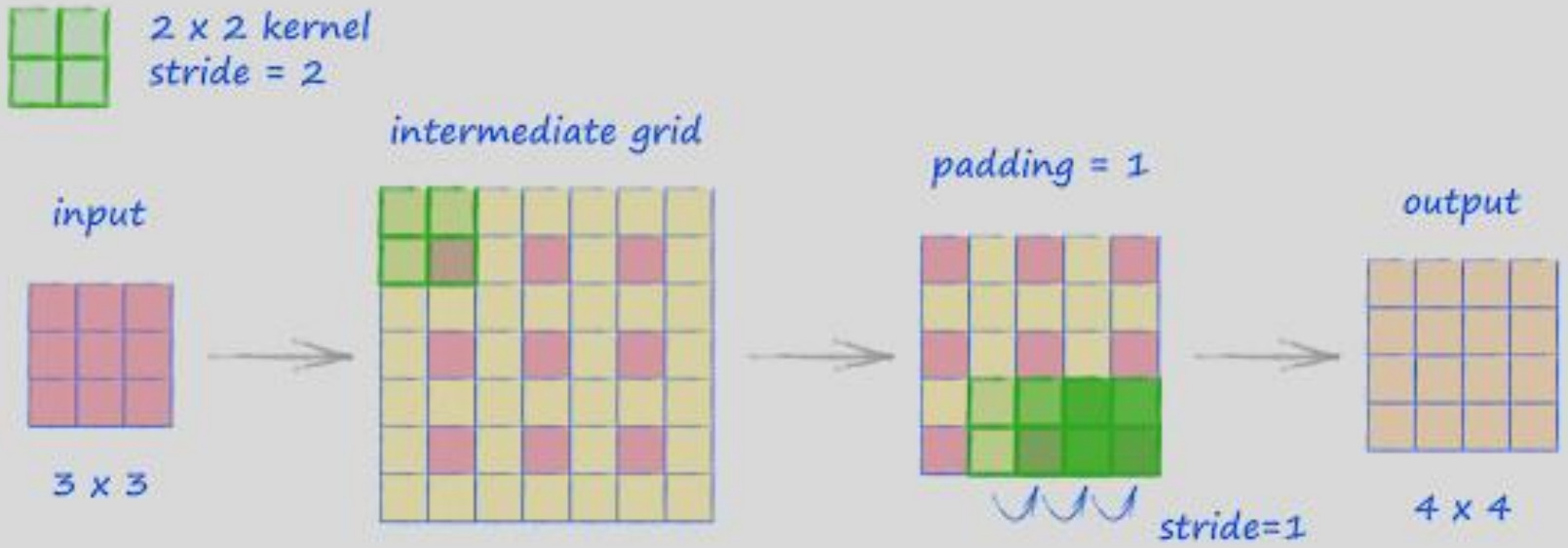
Transposed Convolution with 0 padding, stride 2, 2x2 kernel: Output_size = (input_size-1)*stride – 2*padding + kernel_size + output_padding




```
nn.ConvTranspose2d(in_channels, out_channels, kernel_size=2, stride=2)
```

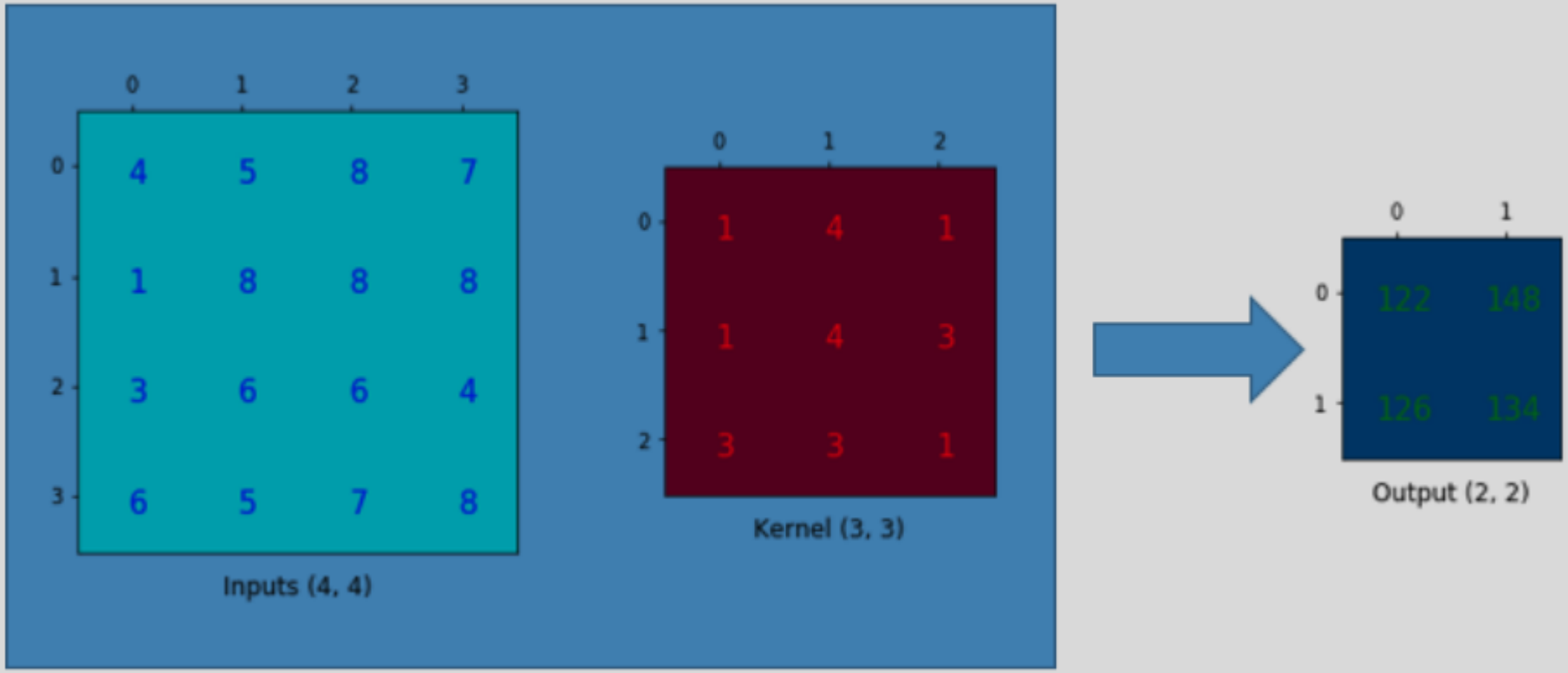
Transposed Convolution

Transposed Convolution with 1 padding, stride 2, 2x2 kernel: Output_size = (input_size-1)*stride – 2*padding + kernel_size + output_padding

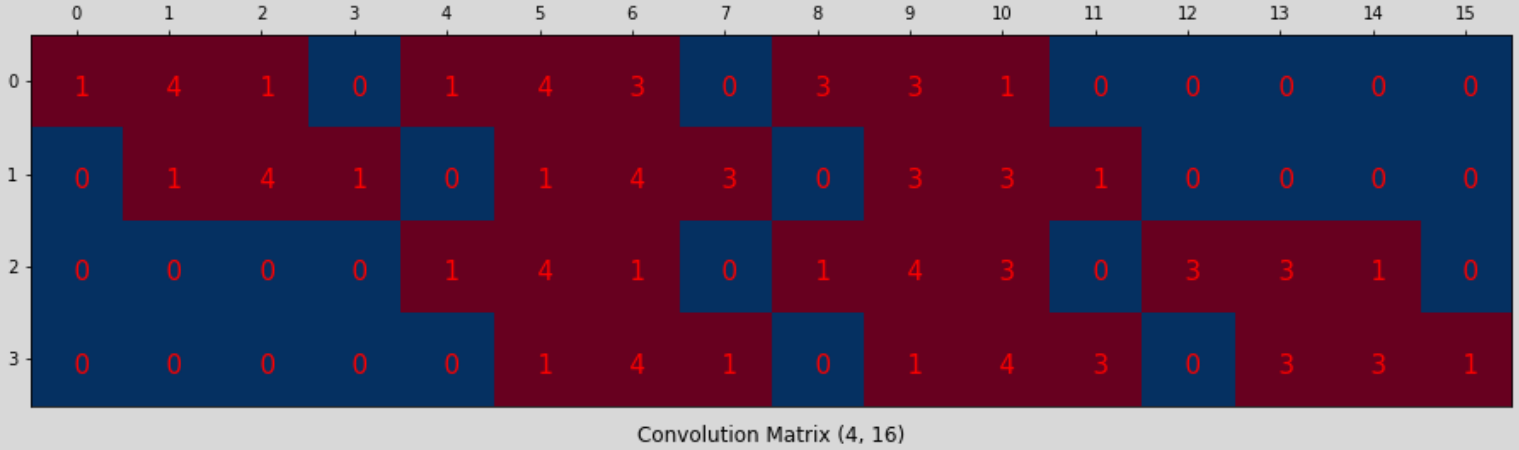



```
nn.ConvTranspose2d(in_channels, out_channels, kernel_size=2, stride=2, padding=1)
```

Why called as Transpose Convoluton?



Why called as Transpose Convoluton?

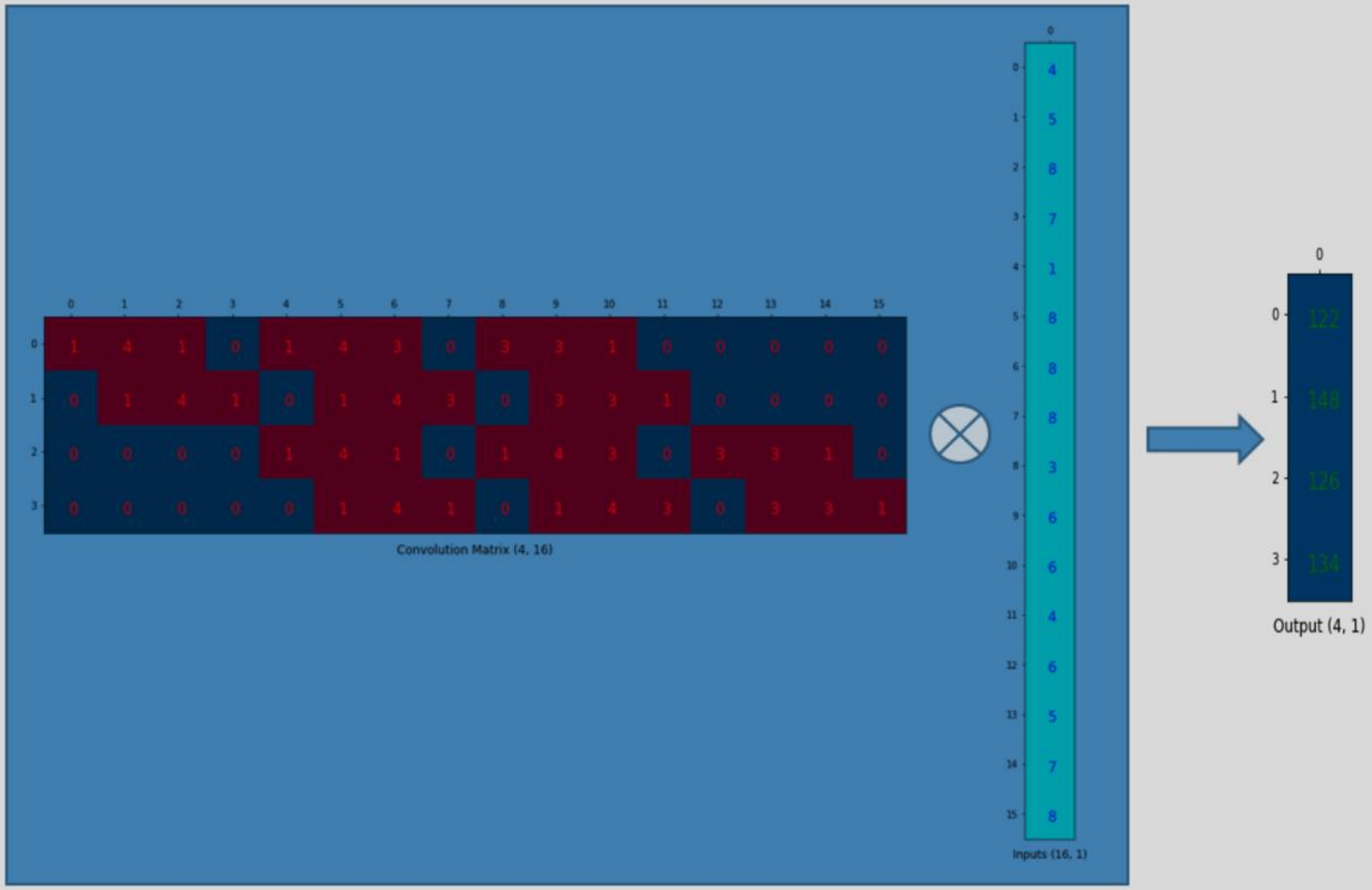


Convolution Matrix (4, 16)

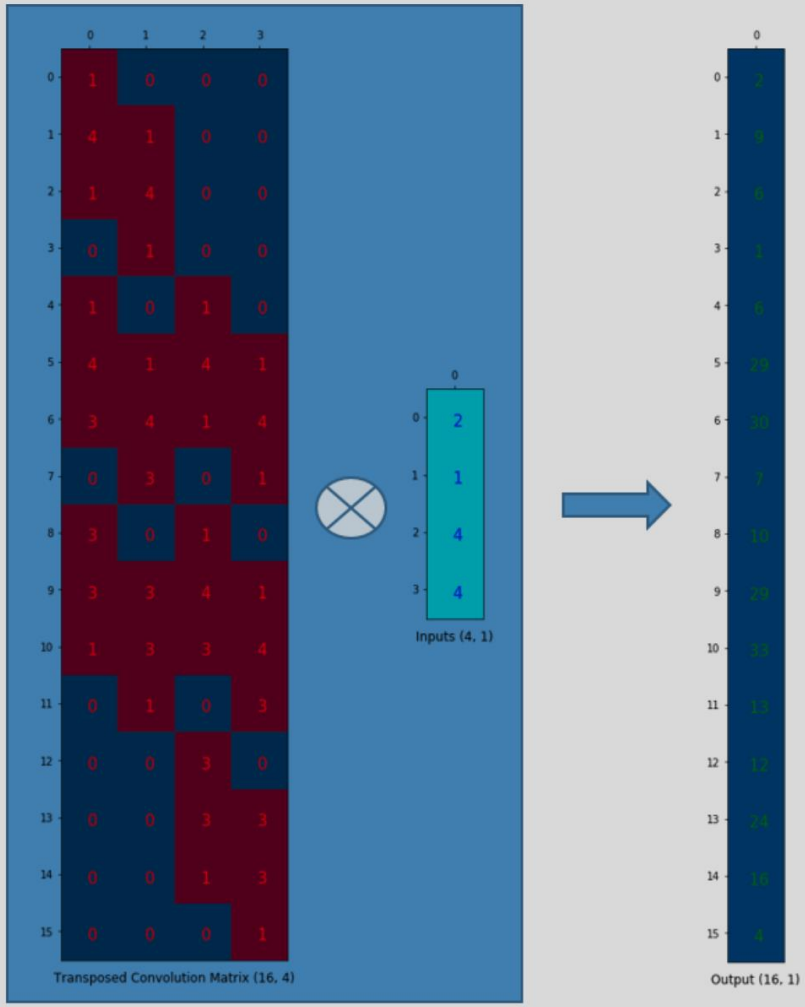
Inputs (4, 4)

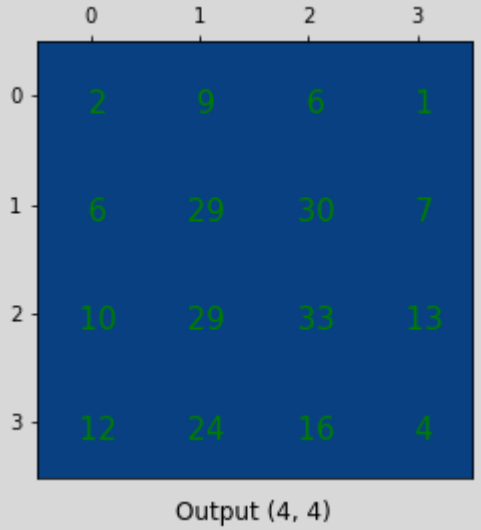
Inputs (16, 1)

Why called as Transpose Convoluton?



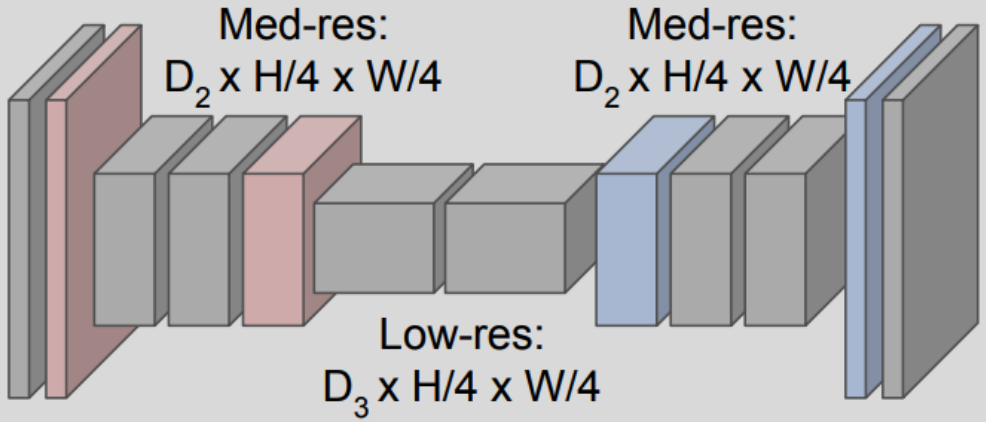
Why called as Transpose Convoluton?





Semantic segmentation

Input:
3 x H x W



Predictions:
H x W

Encoder-decoder

```
class Autoencoder(nn.Module):
    def __init__(self):
        super(Autoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Conv2d(1, 16, 3, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(16, 32, 3, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(32, 64, 7)
        )
        self.decoder = nn.Sequential(
            nn.ConvTranspose2d(64, 32, 7),
            nn.ReLU(),
            nn.ConvTranspose2d(32, 16, 3, stride=2, padding=1, output_padding=1),
            nn.ReLU(),
            nn.ConvTranspose2d(16, 1, 3, stride=2, padding=1, output_padding=1),
            nn.ReLU()
        )

    def forward(self, x):
        x = self.encoder(x)
        x = self.decoder(x)
        return x
```

Output_size = (input_size-1)*stride – 2*padding + kernel_size + output_padding


Thank you!
ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY
2 0 0 7