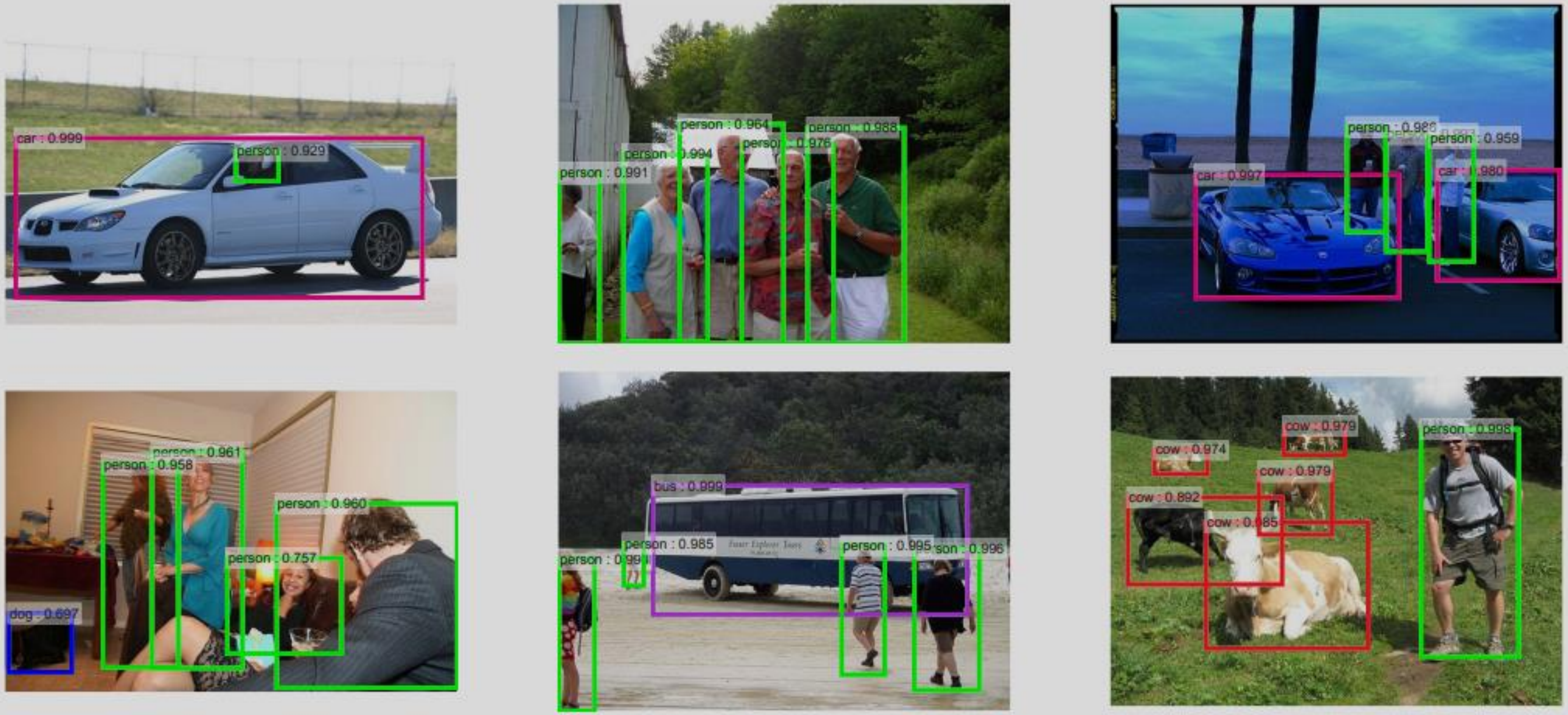
Computer Vision

Lecture 06: Computer Vision Applications

Computer vision applications

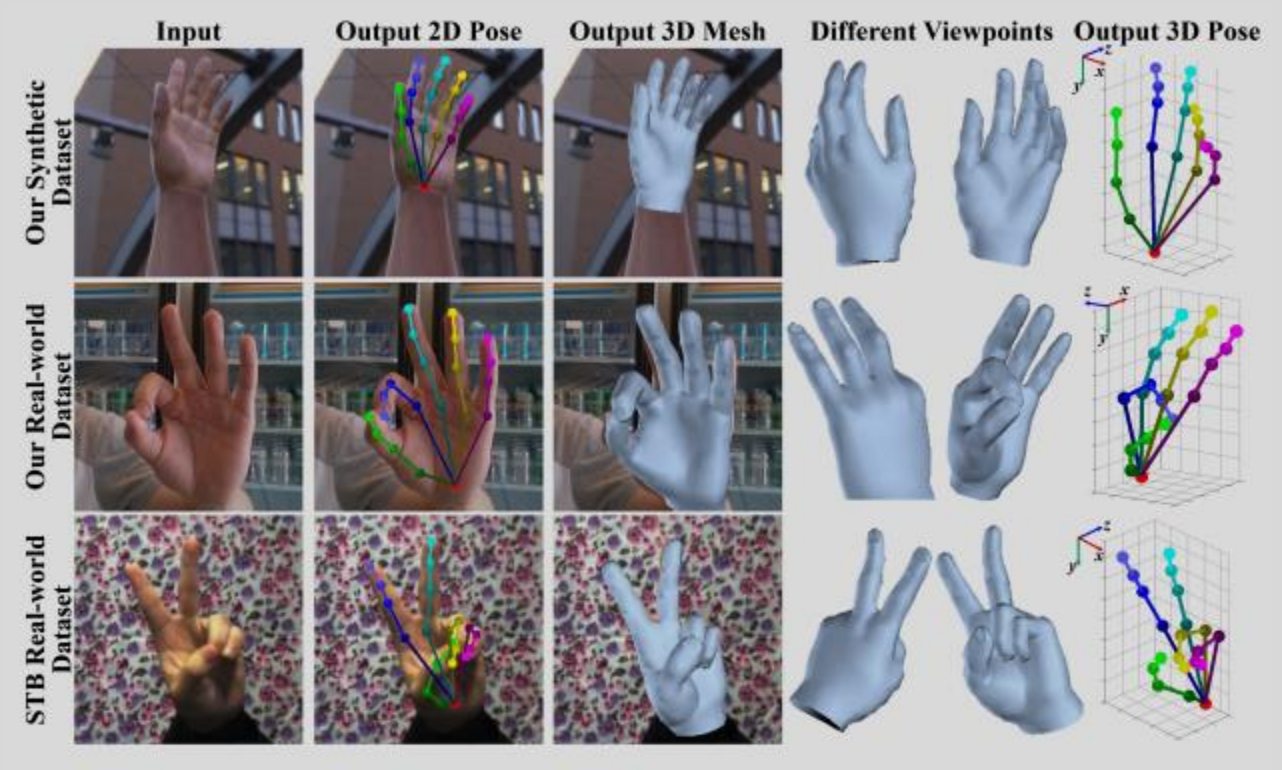


Detecting object locations. [Faster-RCNN NIPS’15]

Computer vision applications

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

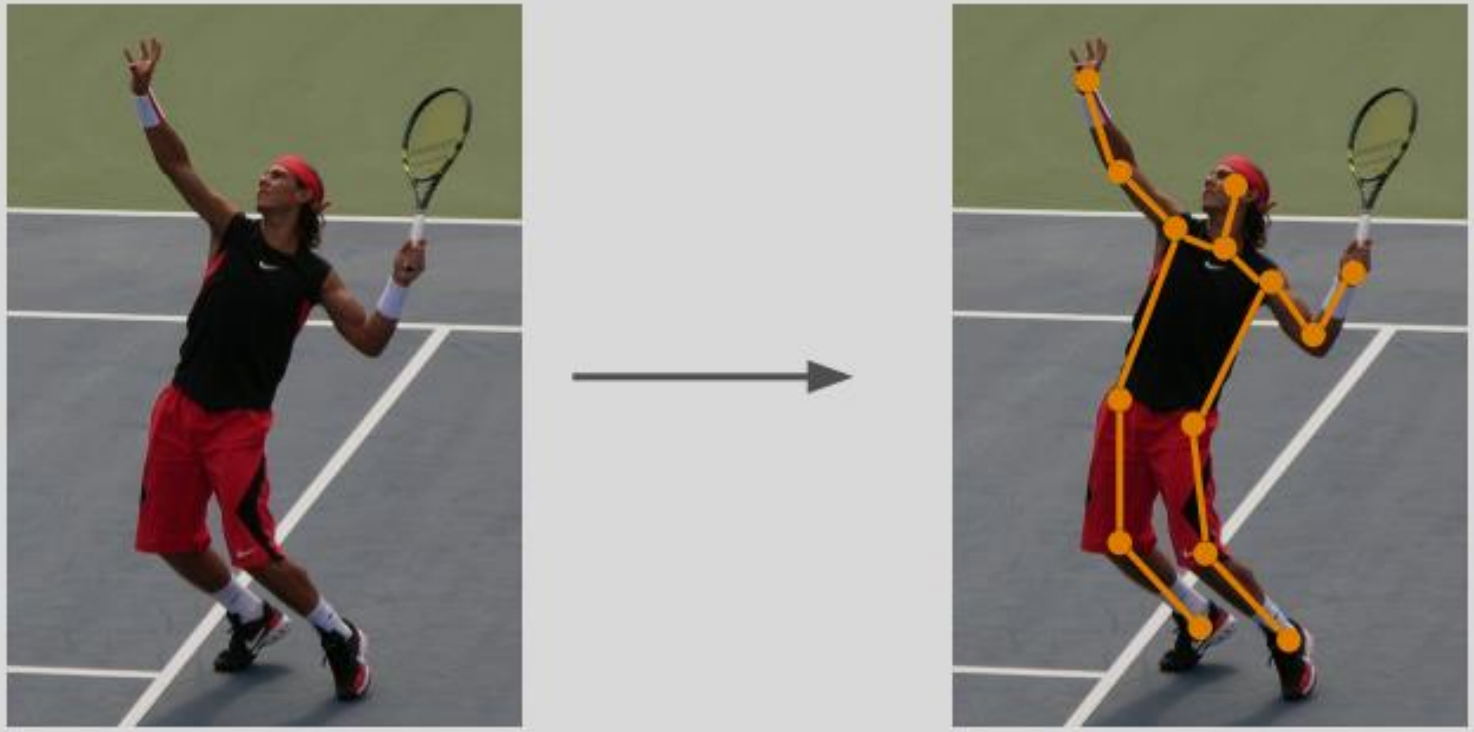
Computer vision applications



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

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

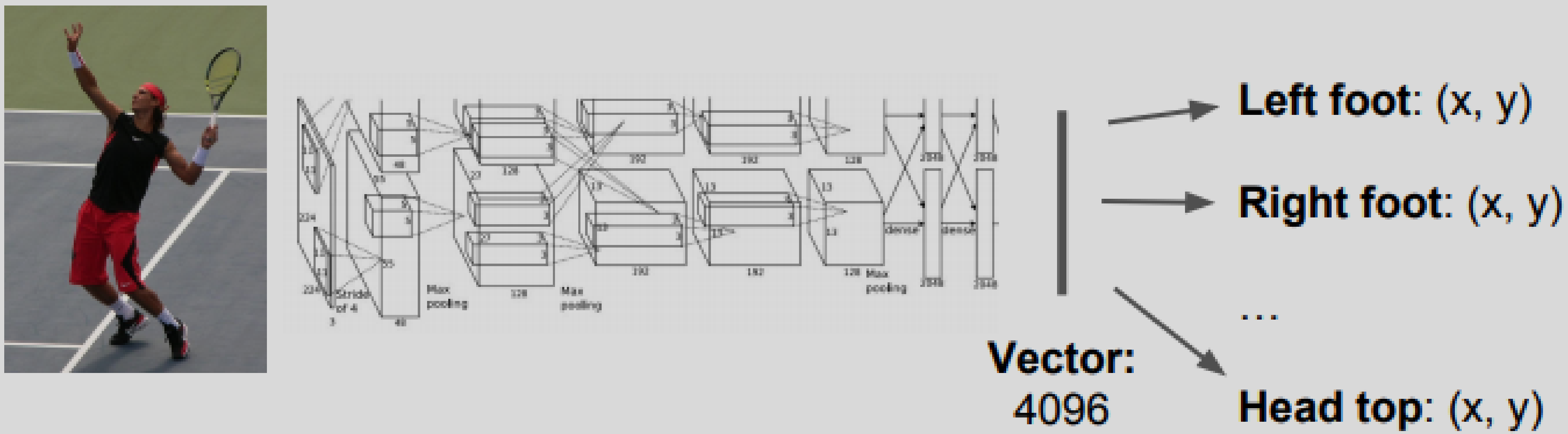
Pose estimation



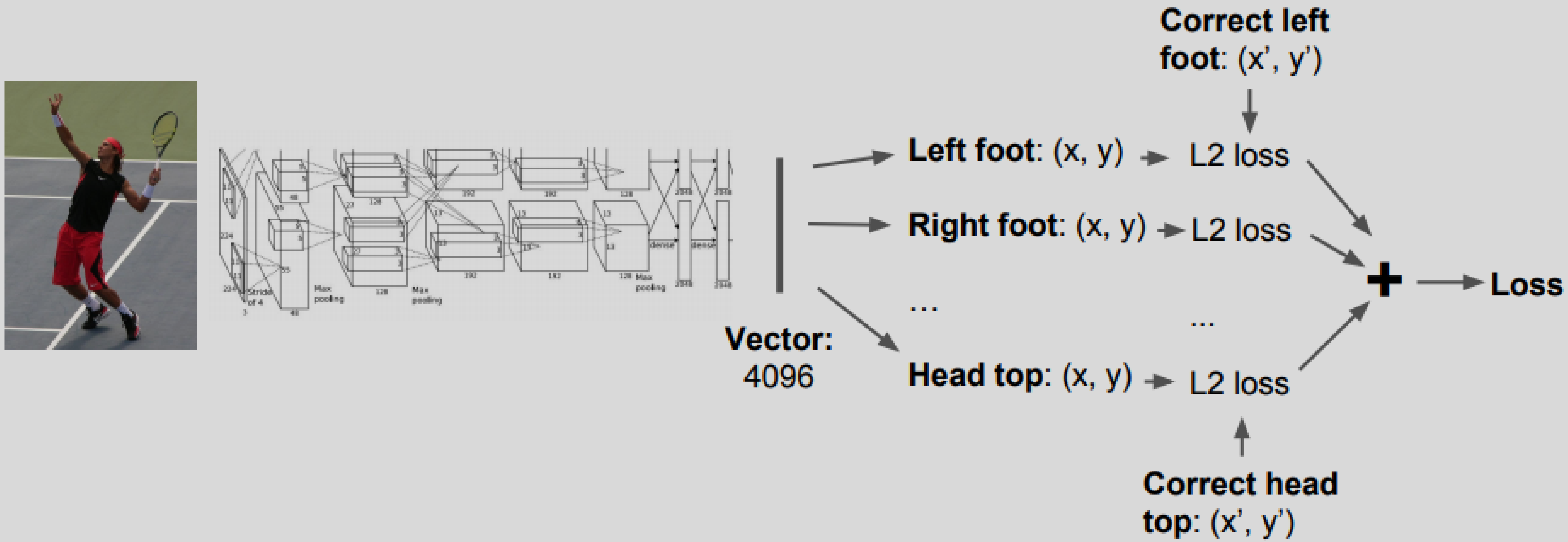
Represent pose as a set of 14 joint positions:

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

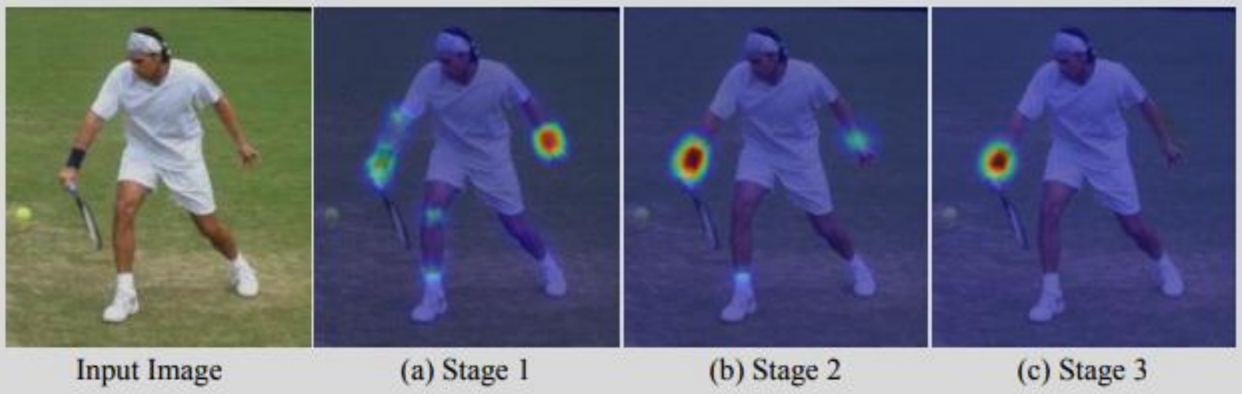
Pose estimation

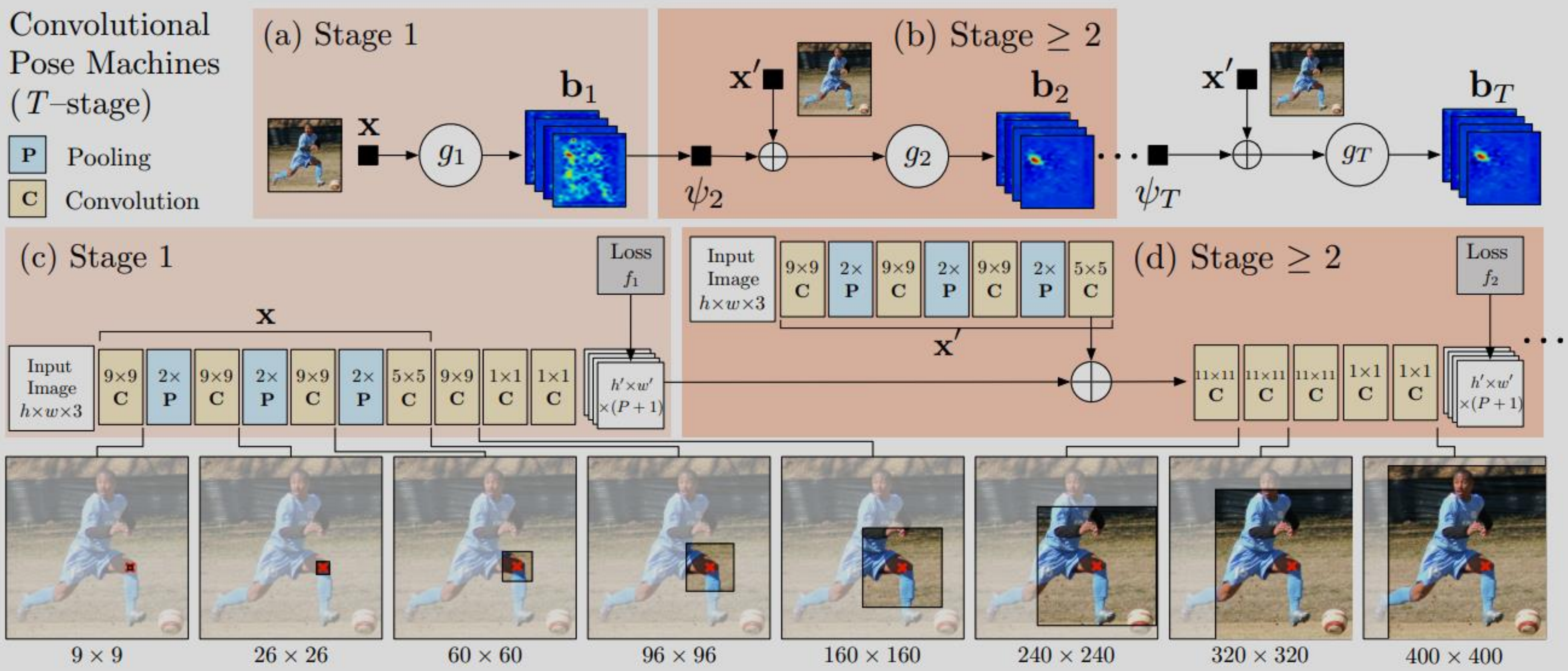


Pose estimation



Pose estimation





Pose estimation

```python
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)
```

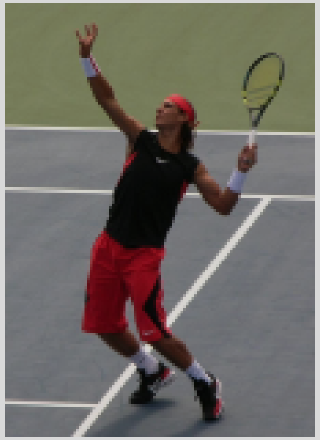
```python
def forward(self, x):
    x = self.relu(self.conv1_1(x))
    x = self.relu(self.conv1_2(x))
    x = self.maxpool(x)
    x = self.relu(self.conv2_1(x))
    x = self.relu(self.conv2_2(x))
    x = self.maxpool(x)
    x = self.relu(self.conv3_1(x))
    x = self.relu(self.conv3_2(x))
    x = self.relu(self.conv3_3(x))
    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
```

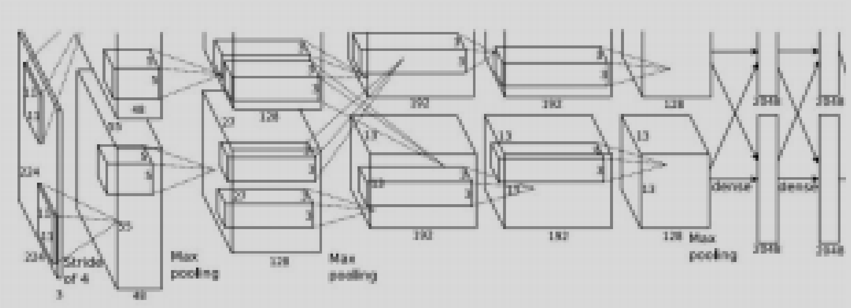
Pose estimation

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





Vector: 4096

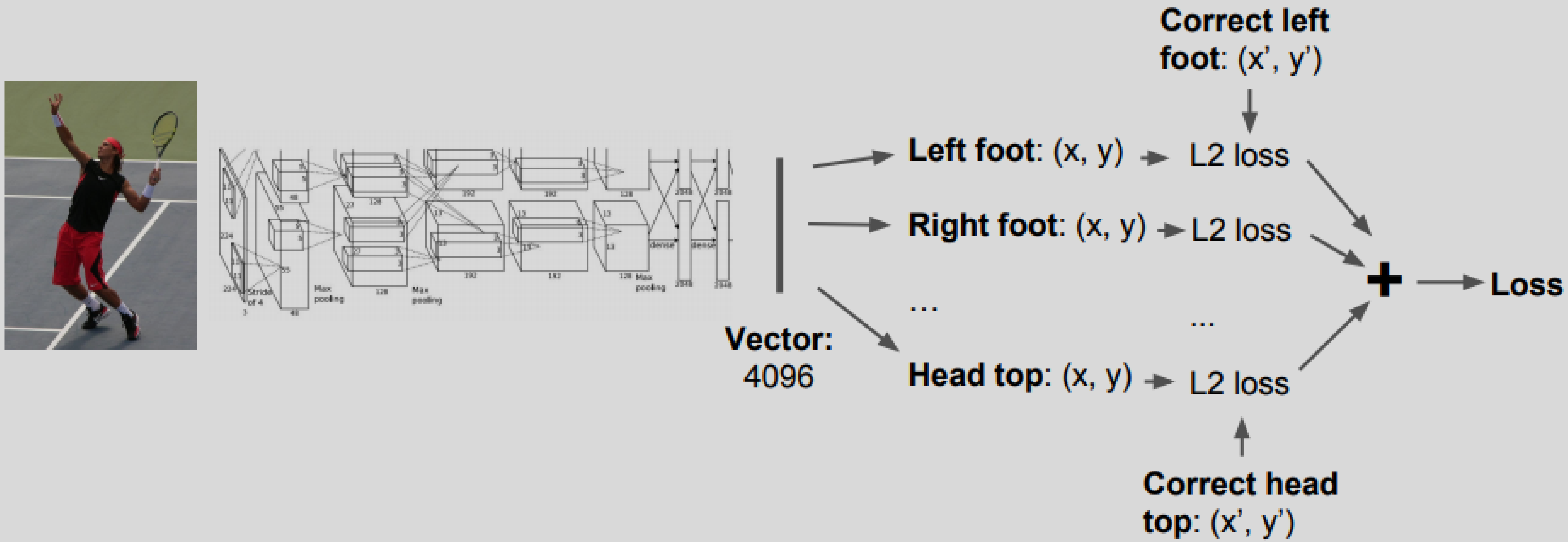
Left foot: (x, y)

Right foot: (x, y)

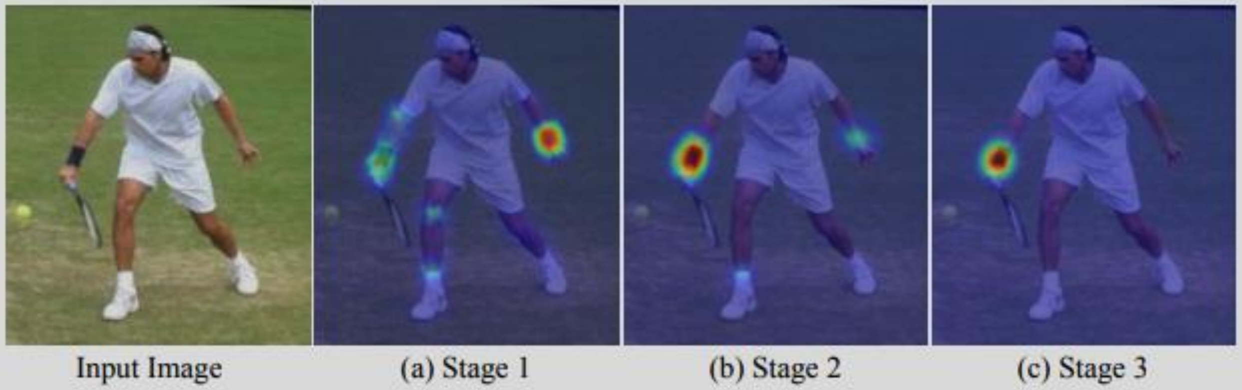
…

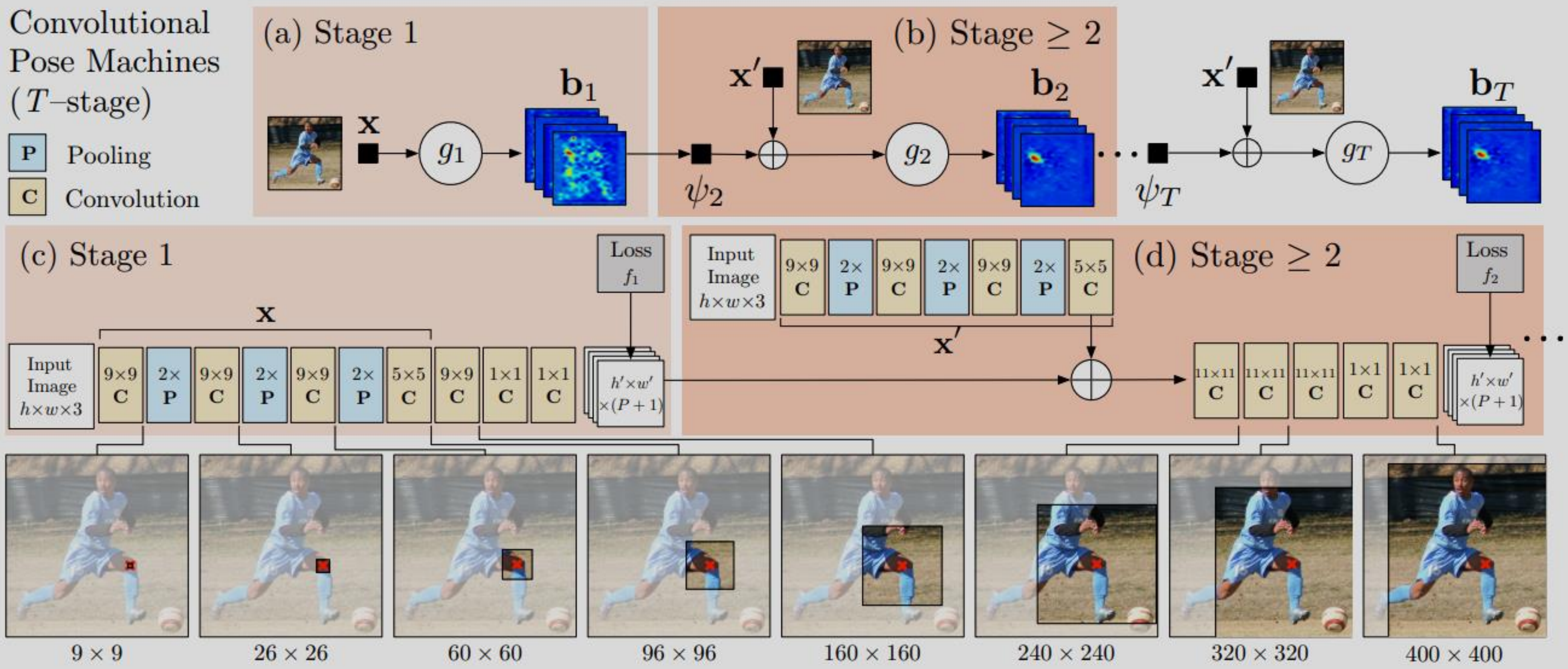
Head top: (x, y)

Pose estimation



Pose estimation



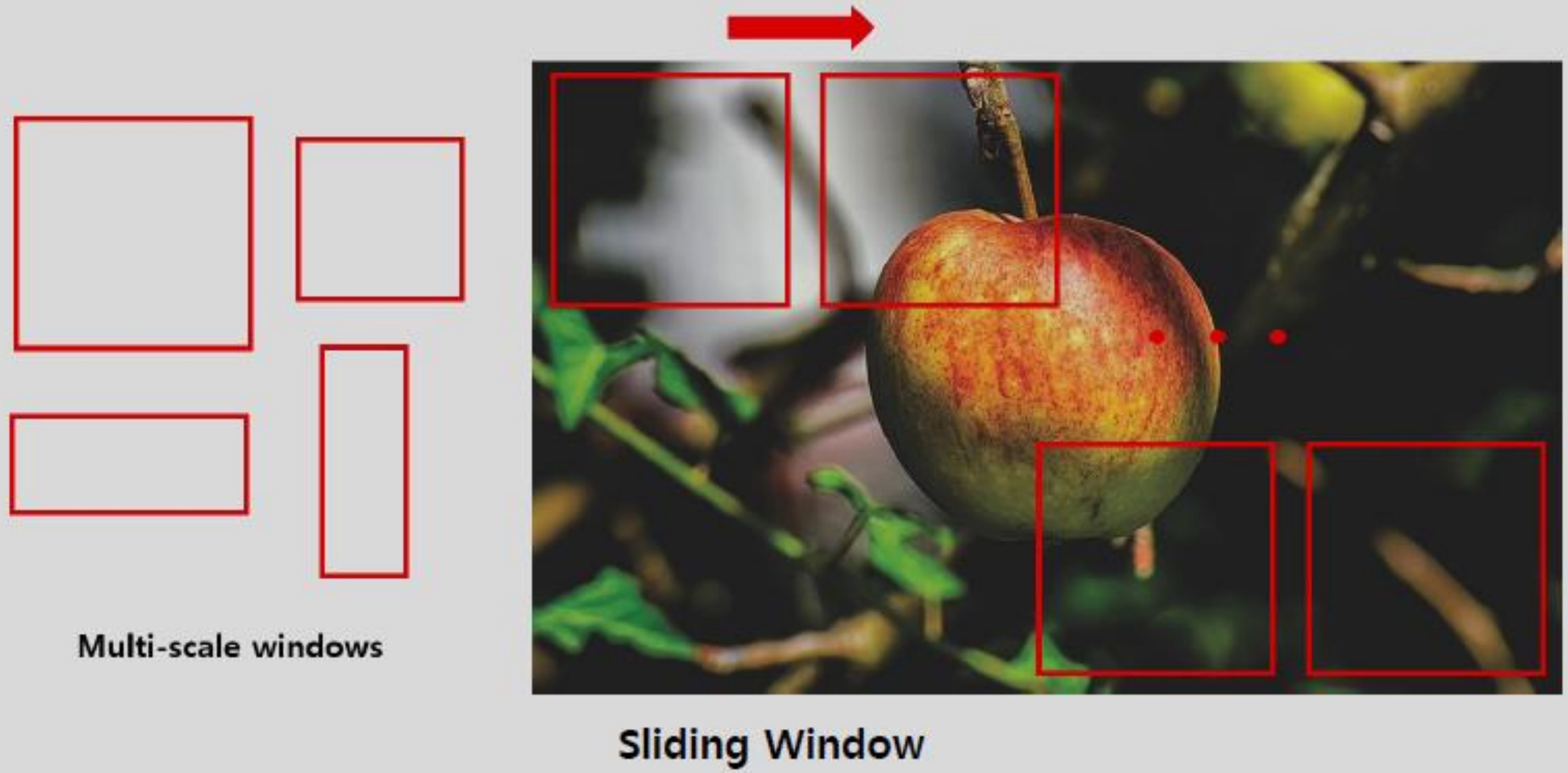


Pose estimation

```python
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)
```

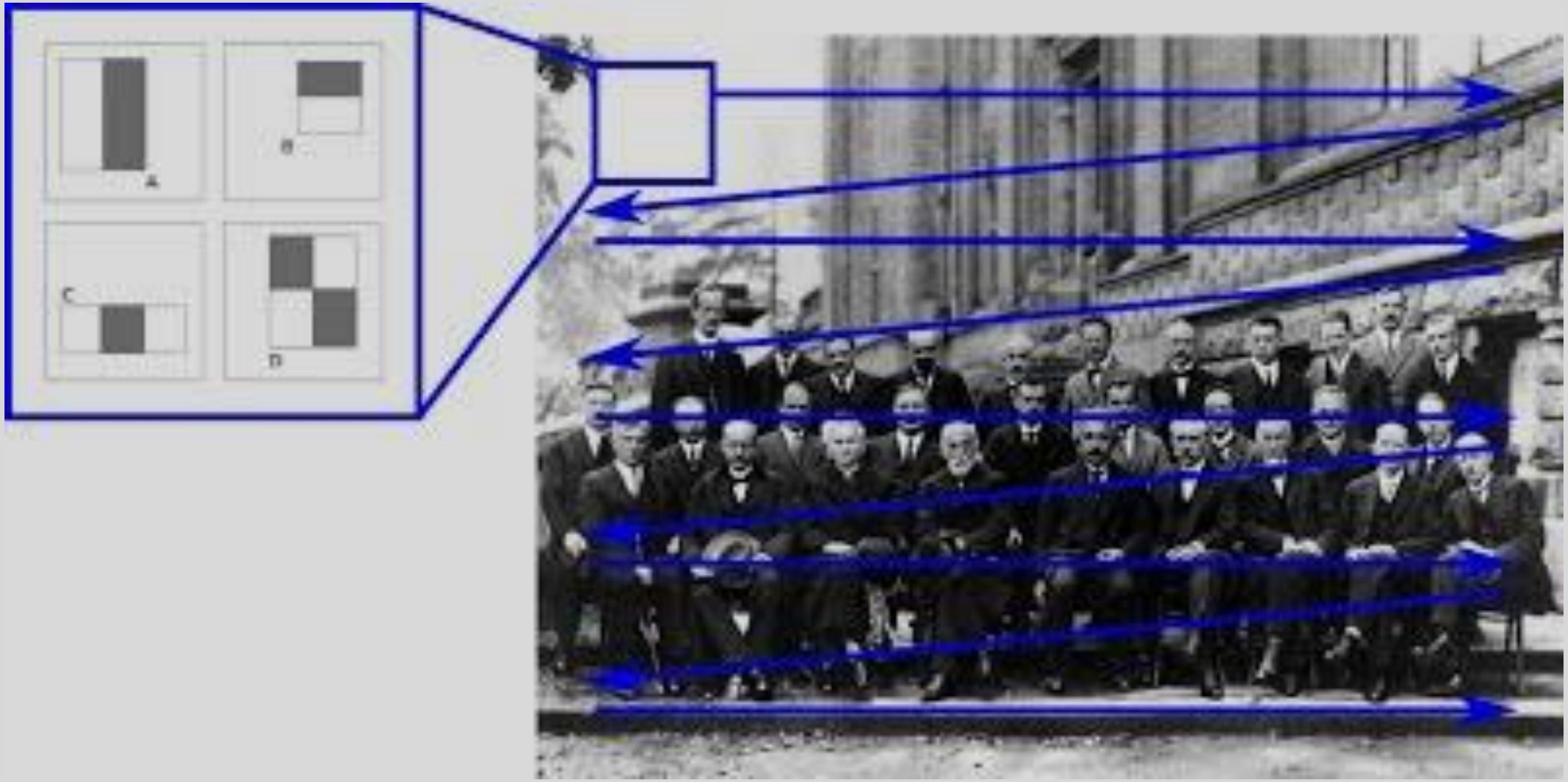
```python
def forward(self, x):
    x = self.relu(self.conv1_1(x))
    x = self.relu(self.conv1_2(x))
    x = self.maxpool(x)
    x = self.relu(self.conv2_1(x))
    x = self.relu(self.conv2_2(x))
    x = self.maxpool(x)
    x = self.relu(self.conv3_1(x))
    x = self.relu(self.conv3_2(x))
    x = self.relu(self.conv3_3(x))
    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



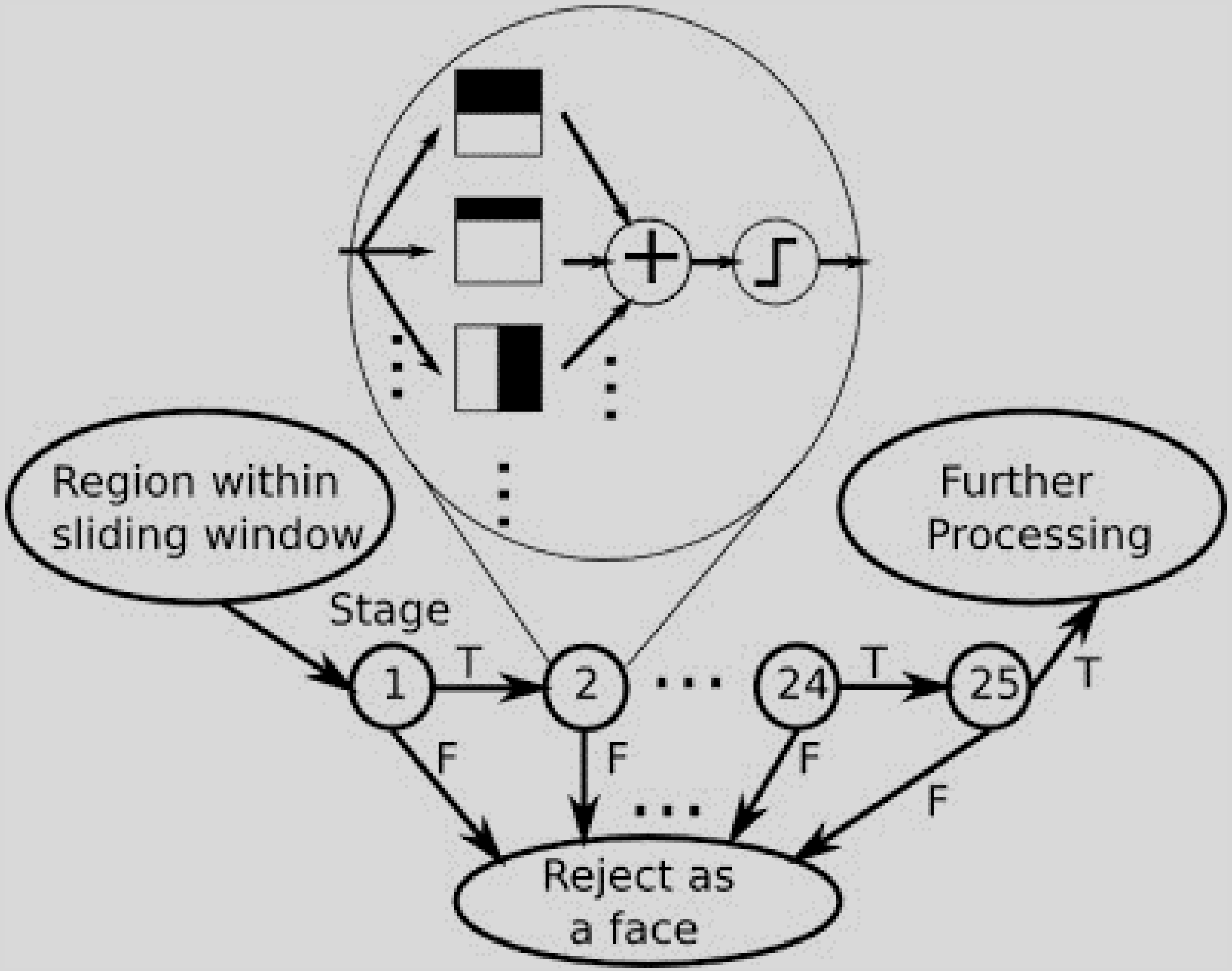
Considering the multi-scales, sliding window is too slow.

Face detection (2001)



Viola & Jones, CVPR'01

Face detection (2001)

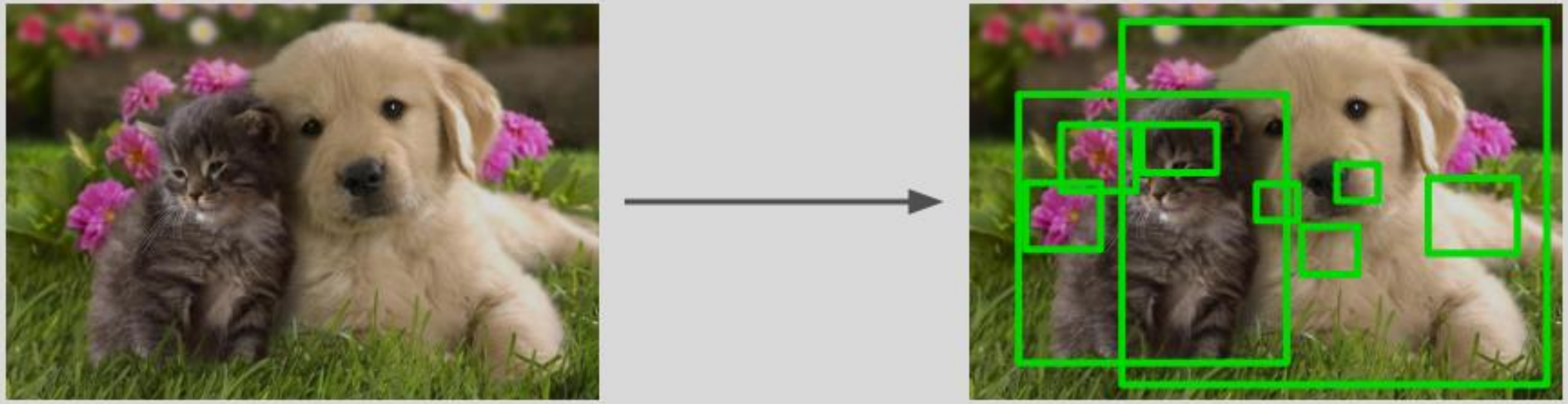


Viola & Jones, CVPR'01

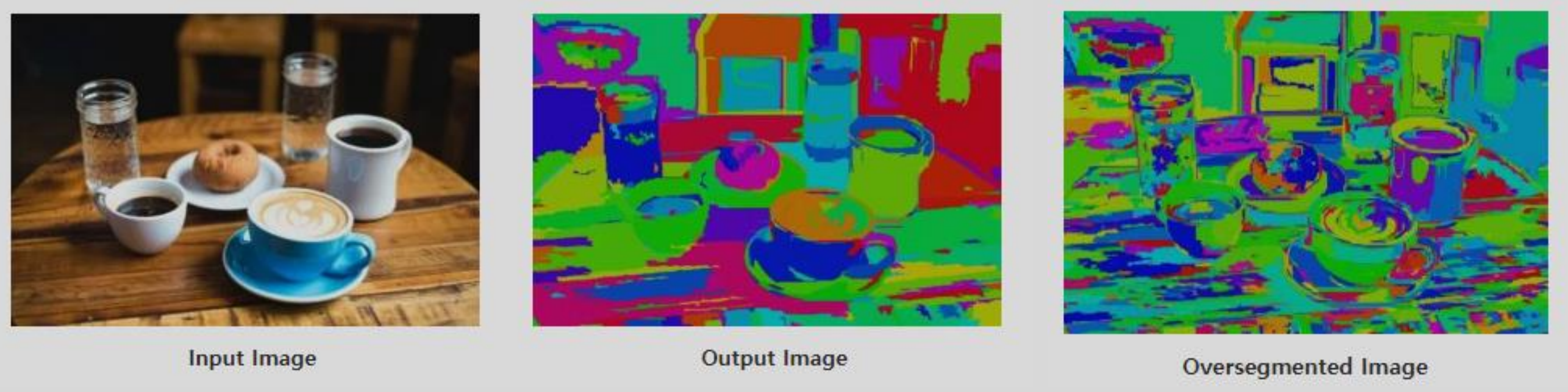
Object proposal

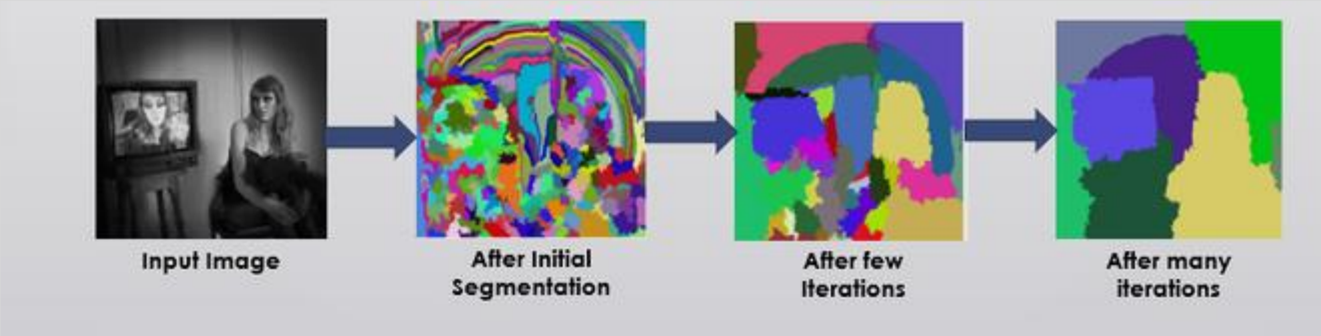
Find image regions that are likely to contain objects.

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



Object proposal





Considering Color, Texture, Size, Shape similarities.

Object proposal

Algorithm 1: Hierarchical Grouping Algorithm

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

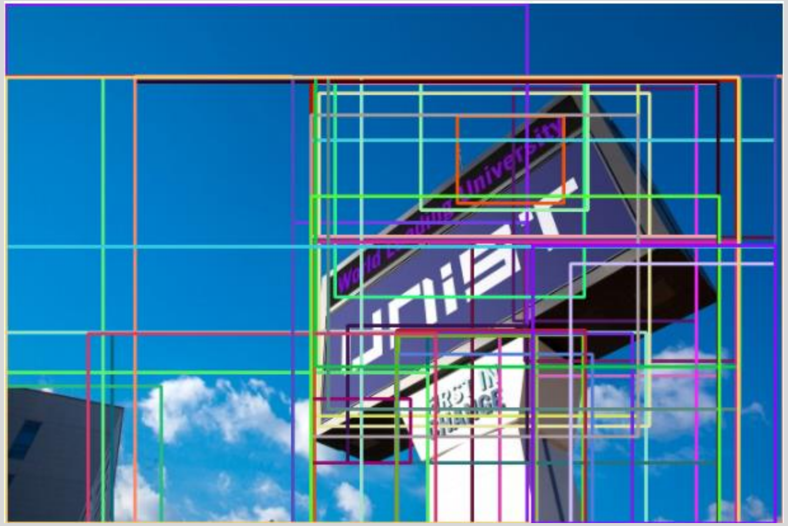
Obtain initial regions $R = \{r_1, \cdots, r_n\}$ using [13]
Initialise similarity set $S = \emptyset$
foreach *Neighbouring region pair* (r_i, r_j) **do**
    Calculate similarity $s(r_i, r_j)$
    $S = S \cup s(r_i, r_j)$

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)
```



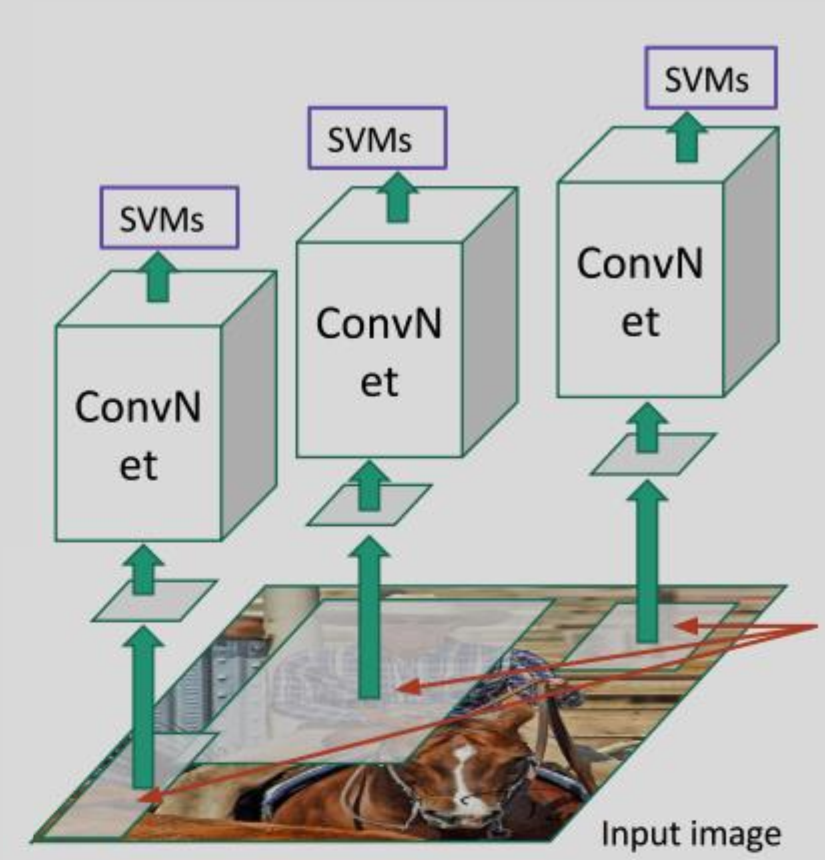
R-CNN (CVPR'13)


Input image

R-CNN (CVPR'13)

Regions of interest
From selective search (~2000).

R-CNN (CVPR'13)



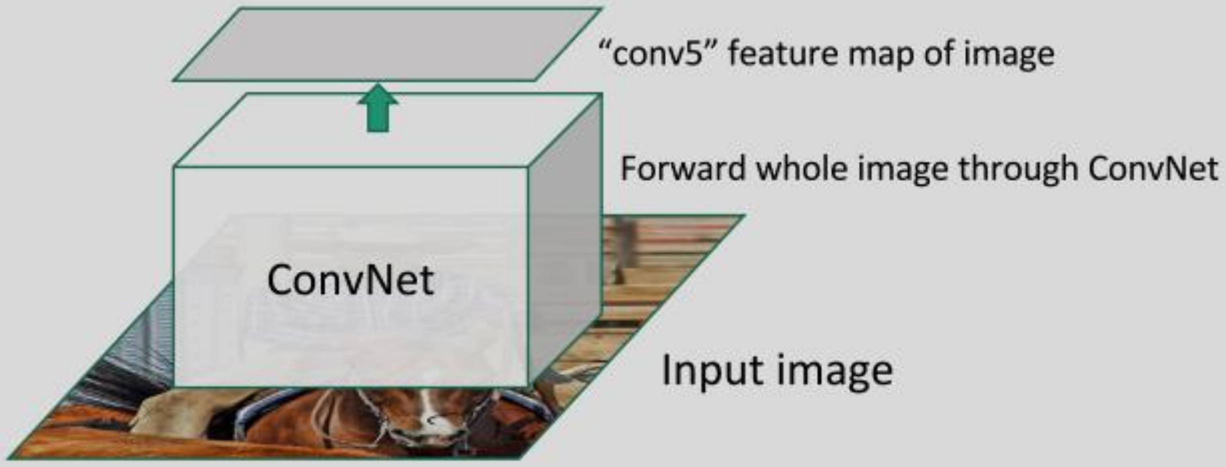
Classify each region with SVMs.

Forward with ImageNet-trained CNNs.

Regions of interest
From selective search (~2000).

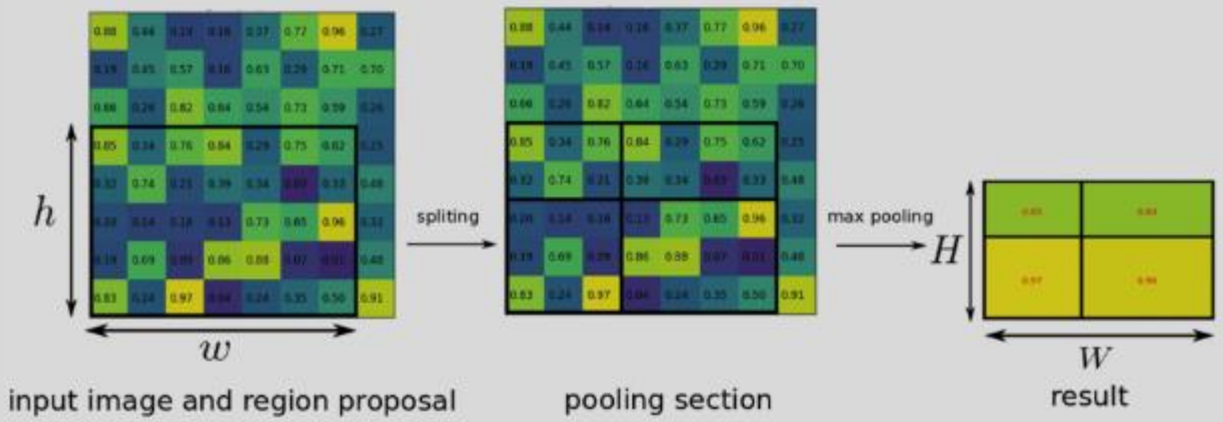
Limitations:
1) Not end-to-end trainable.
2) Still slow.

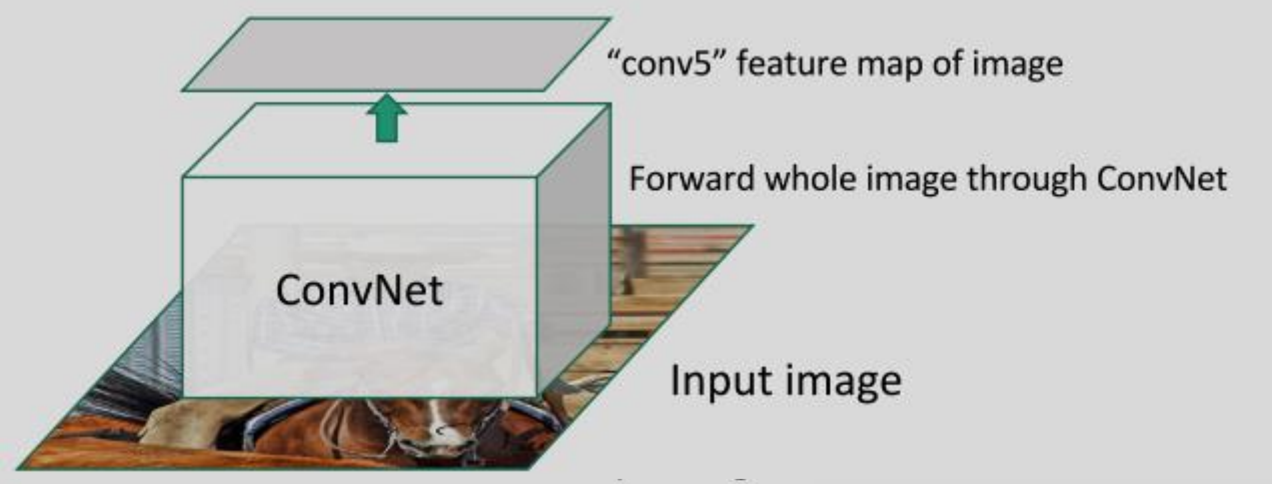
Fast R-CNN (ICCV'15)



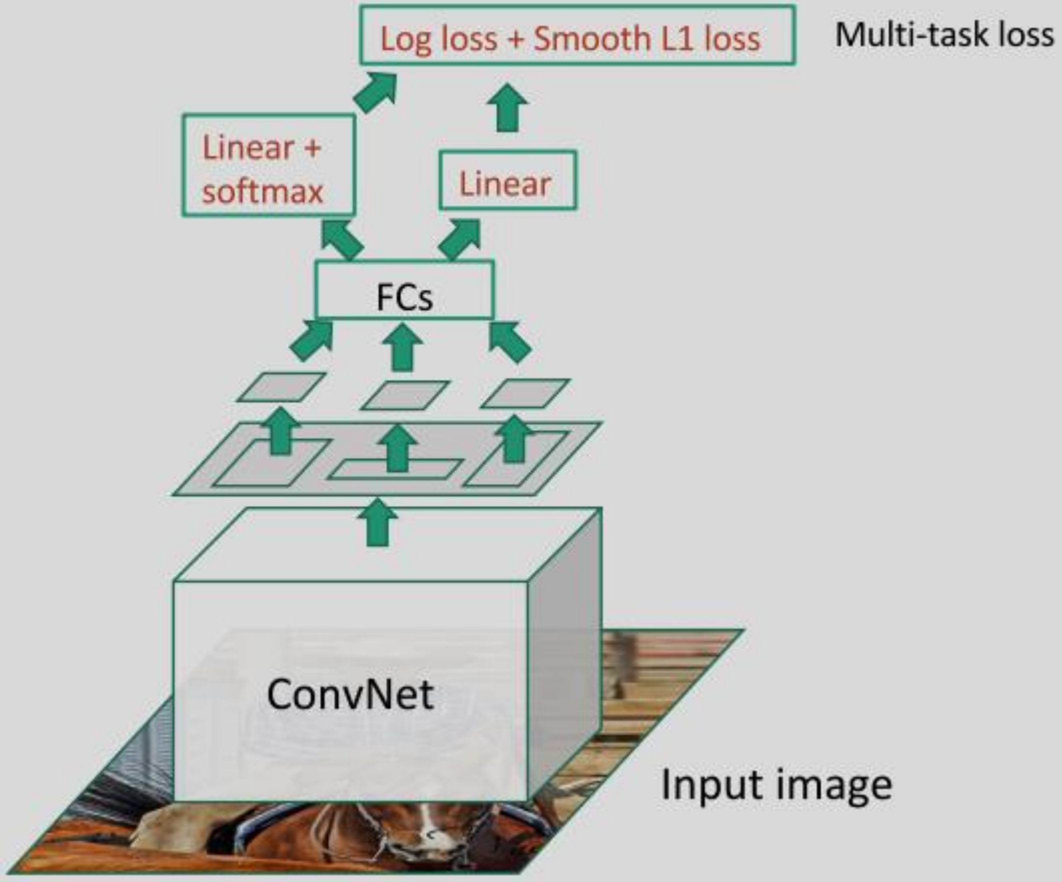
Fast R-CNN (ICCV'15)

RoI Pooling





Fast R-CNN (ICCV'15)



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

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

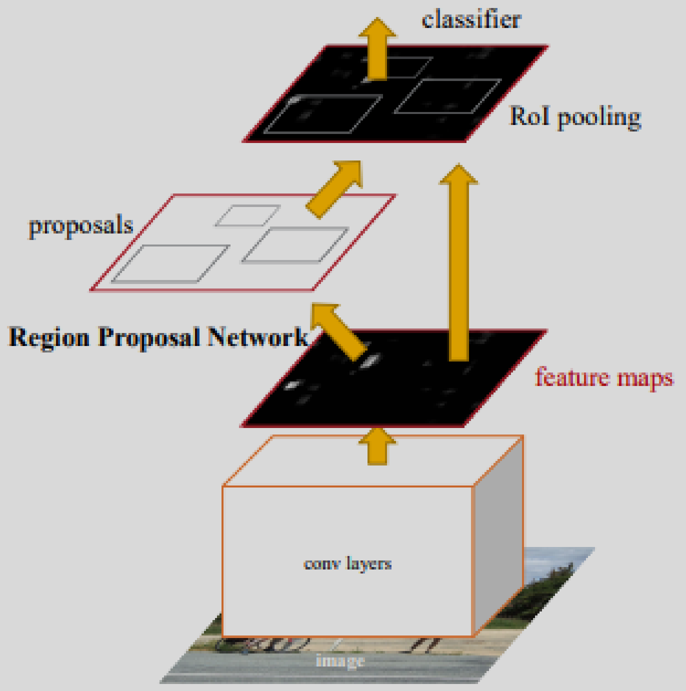
$$t^u = (t^u_{\text{x}}, t^u_{\text{y}}, t^u_{\text{w}}, t^u_{\text{h}})$$

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

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

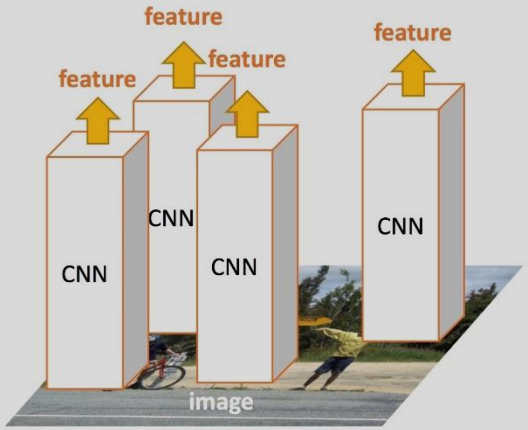
$$\text{smooth}_{L_1}(x) = \begin{cases} 0.5x^2 & \text{if } |x| < 1 \\ |x| - 0.5 & \text{otherwise,} \end{cases}$$

Faster R-CNN (NIPS'15)

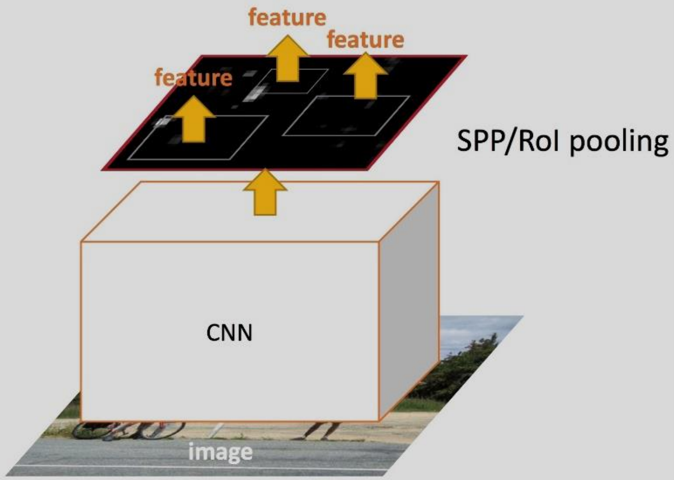


Solve the bottleneck in the region proposal of the Fast-RCNN

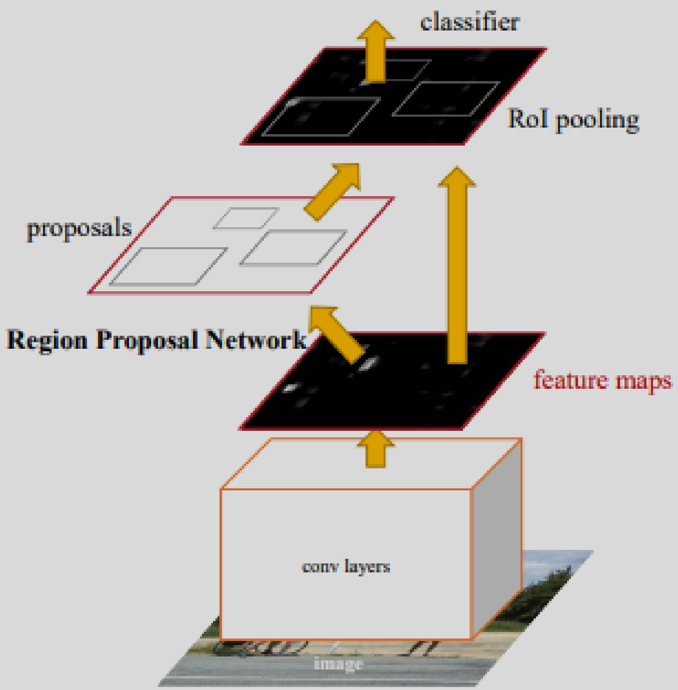
Comparisons



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



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**

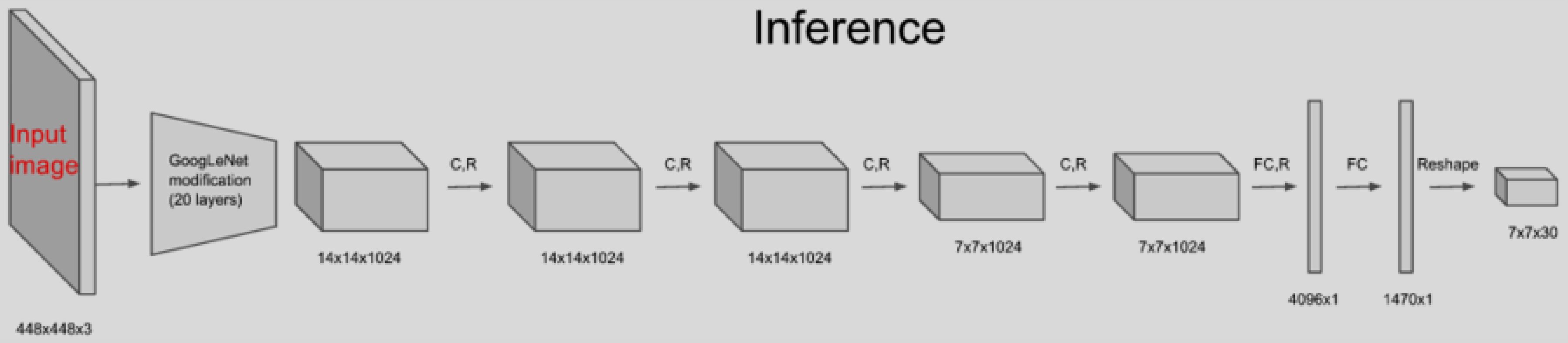
Detection mAP on PASCAL VOC 2007 and 2012, with VGG-16 pre-trained on ImageNet Dataset

YOLO (You only look once, CVPR'16)

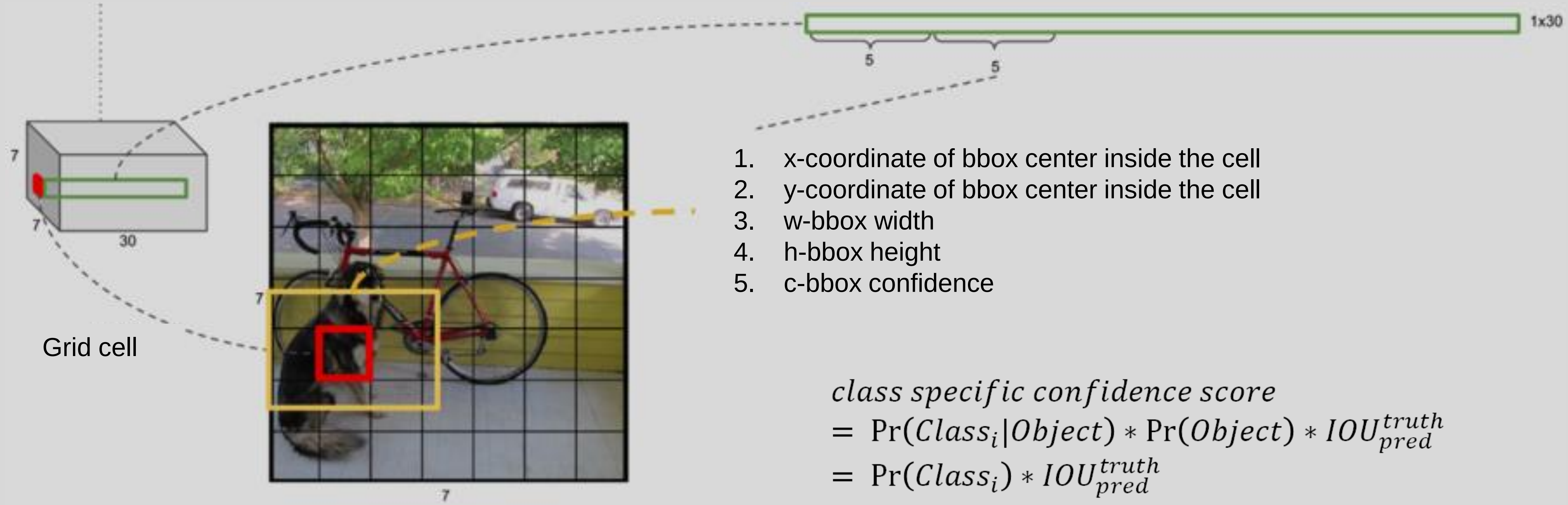
- 1 stage algorithm

YOLO (You only look once, CVPR'16)

- Two candidates for each grid, 20 classes:



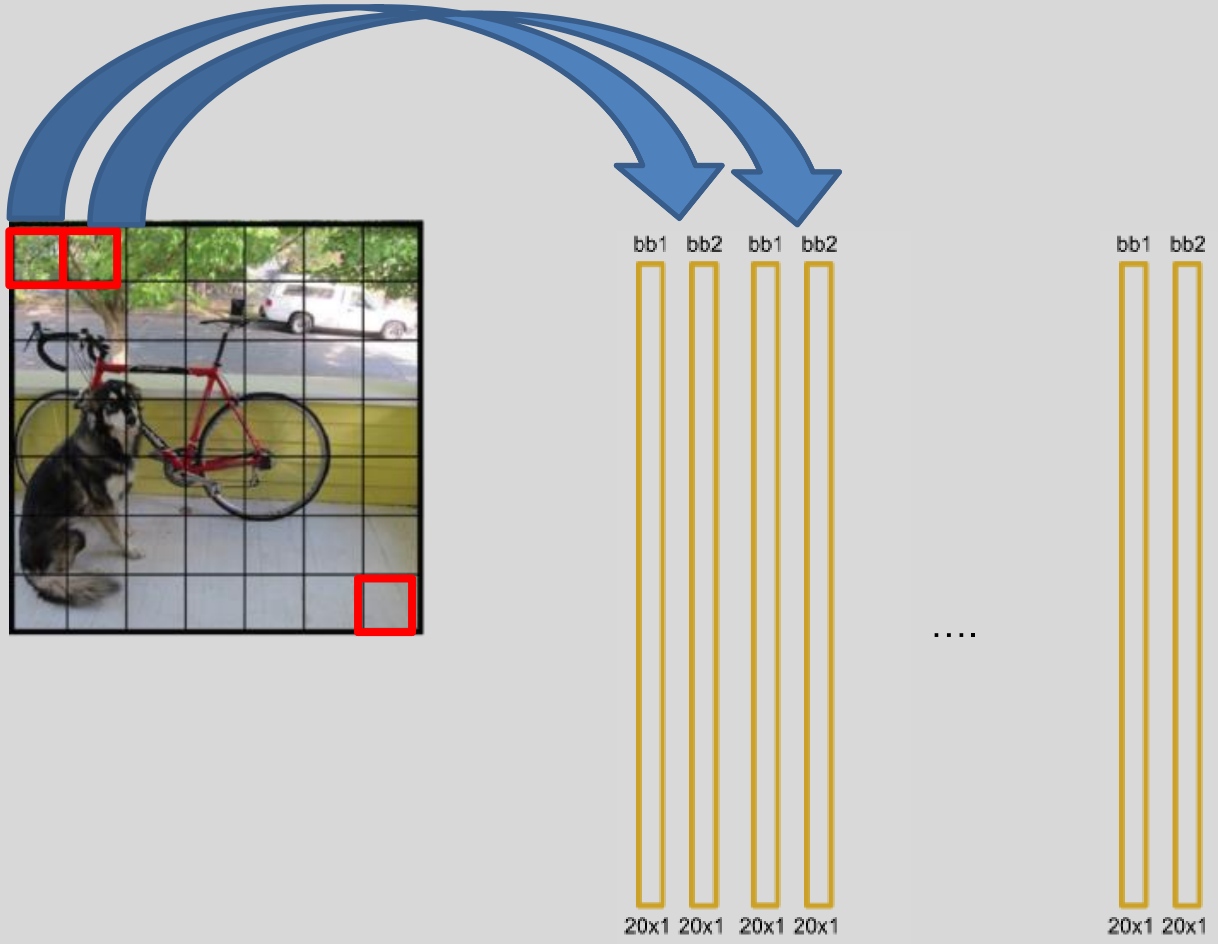
YOLO (You only look once, CVPR’16)



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

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

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
```

Non-maximal suppression

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

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
```

YOLO (You only look once, CVPR’16)

- 1 stage algorithm

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)

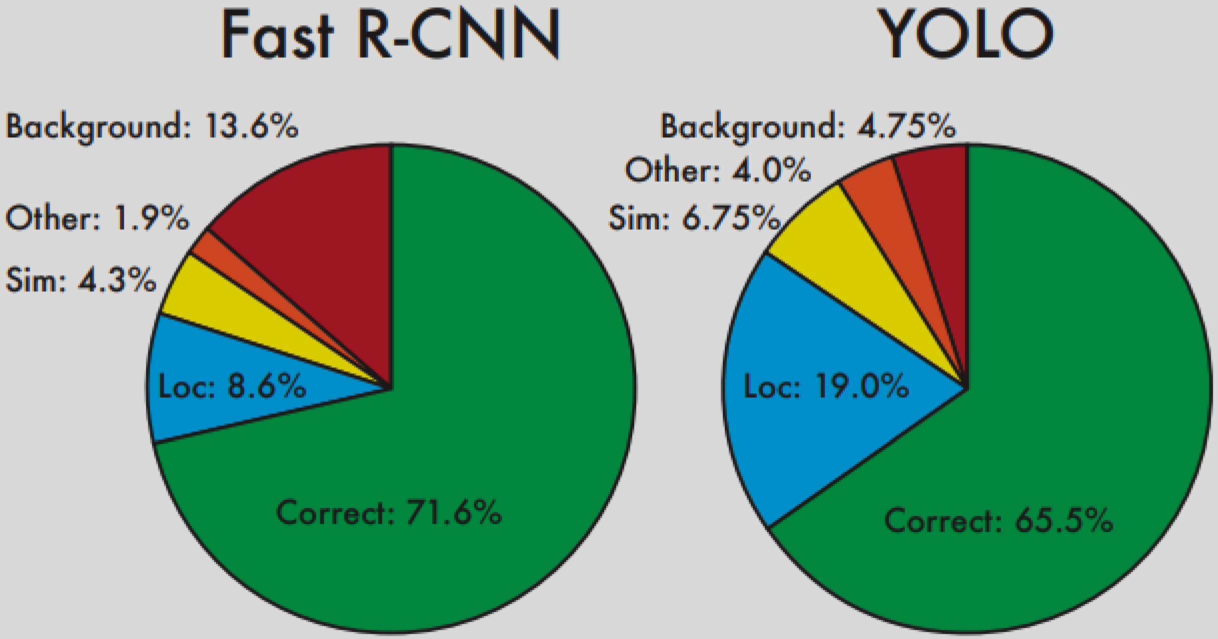
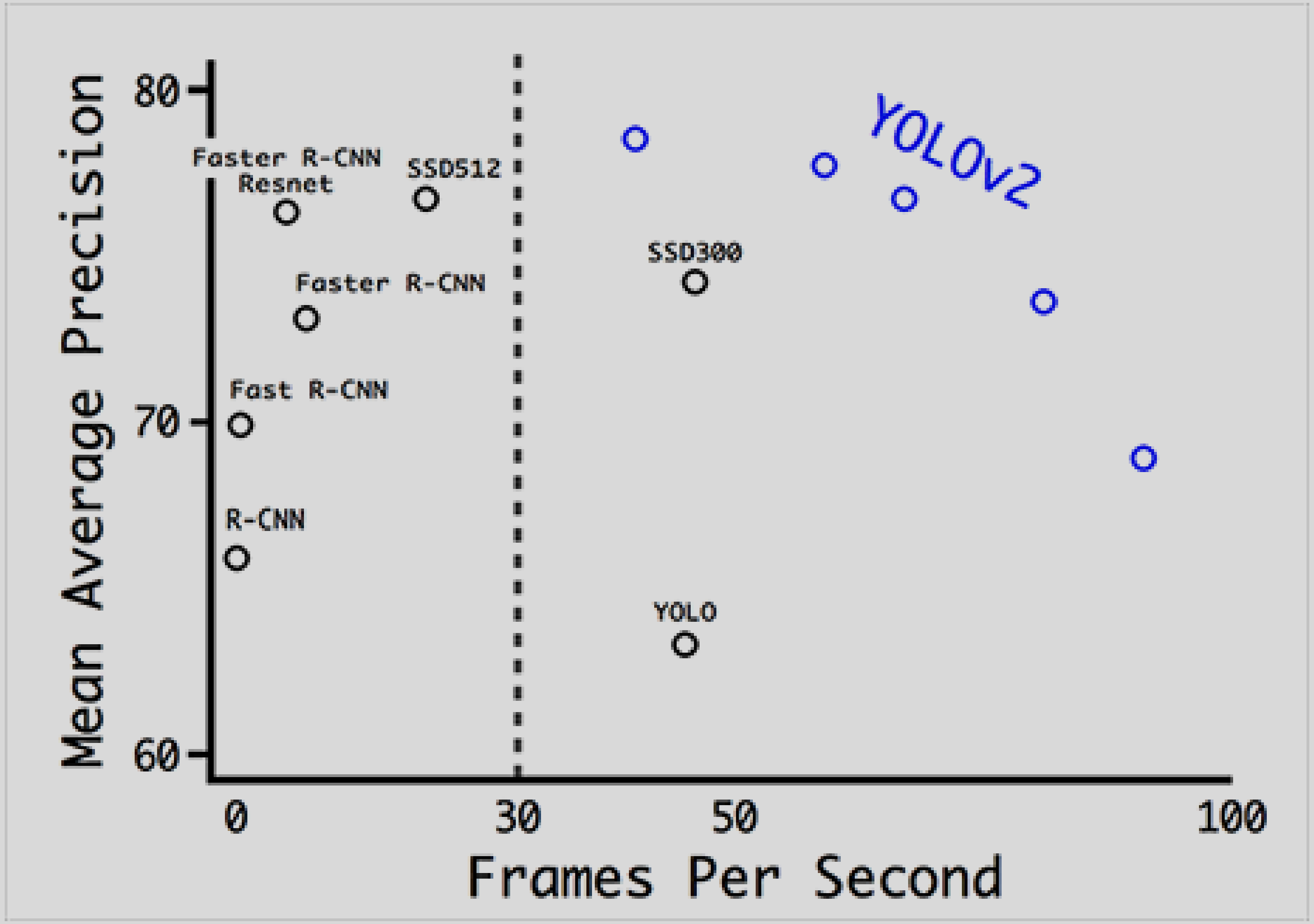


Figure 4: Error Analysis: Fast R-CNN vs. YOLO These charts show the percentage of localization and background errors in the top N detections for various categories (N = # objects in that category).

YOLO (You only look once, CVPR’16)



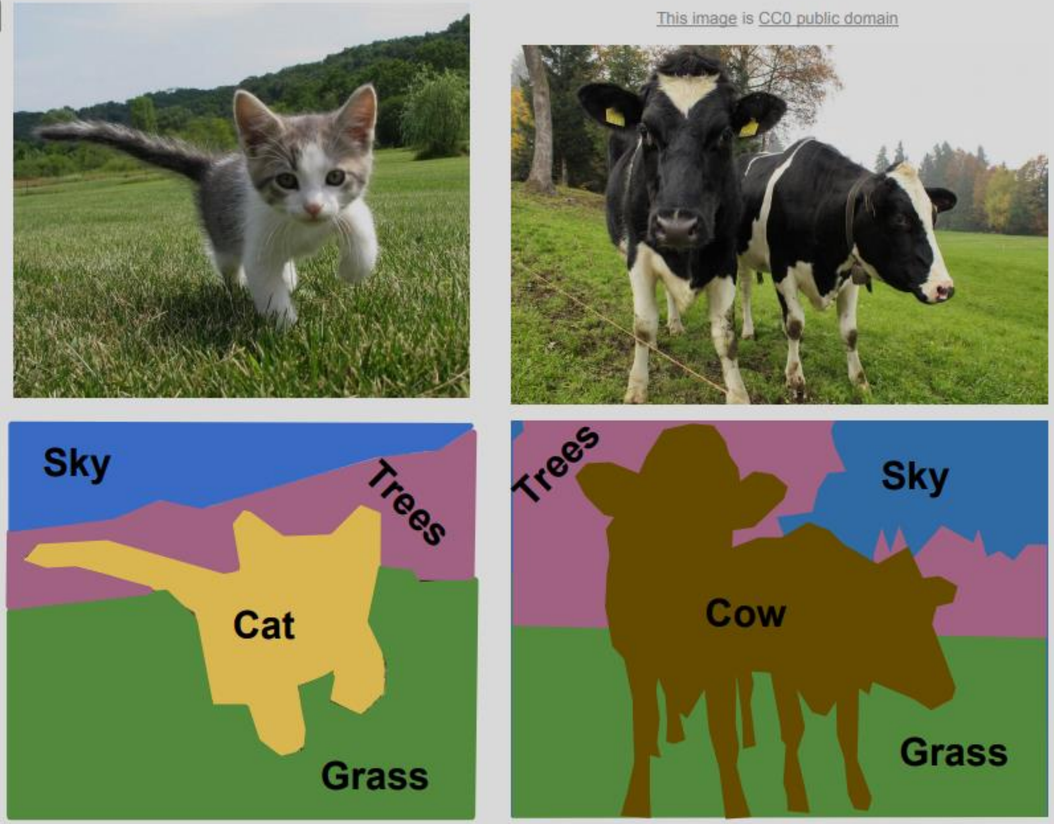
Semantic segmentation

Semantic Segmentation

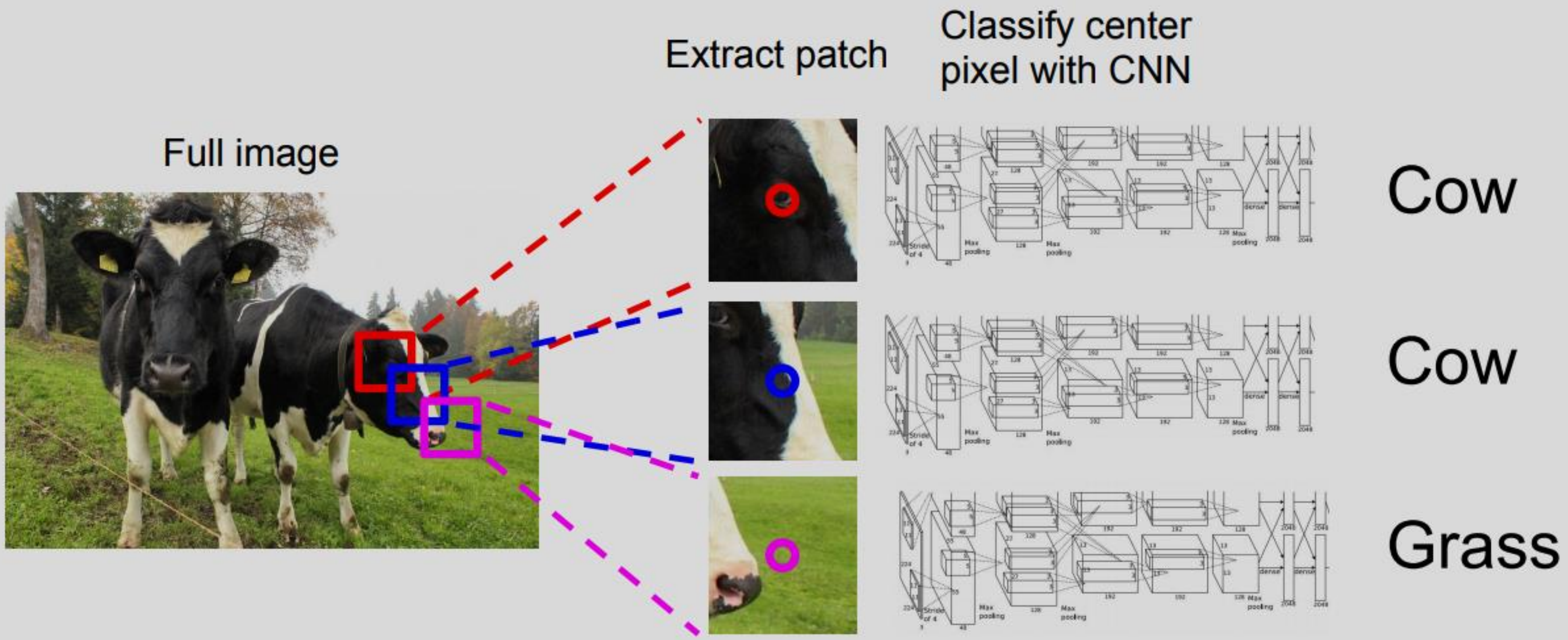
Label each pixel in the image with a category label

Don’t differentiate instances, only care about pixels

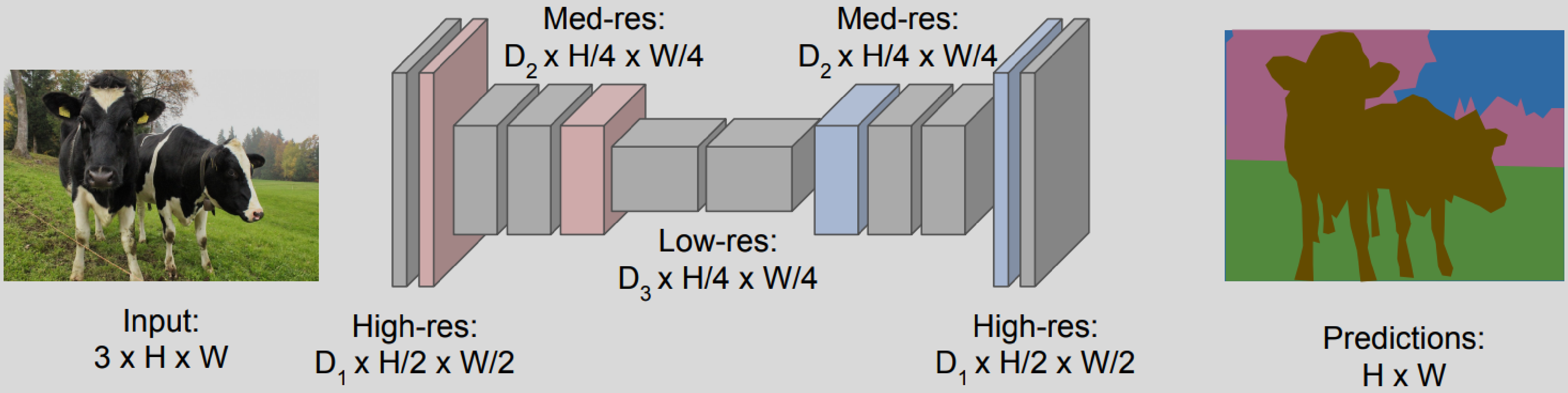
This image is CC0 public domain



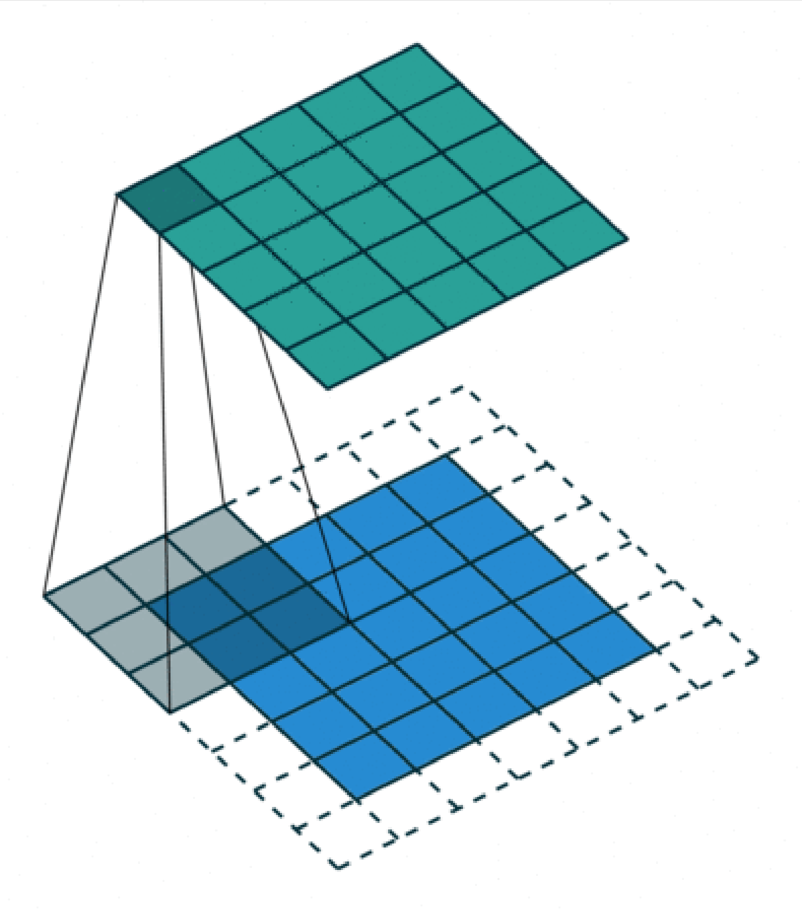
Semantic segmentation



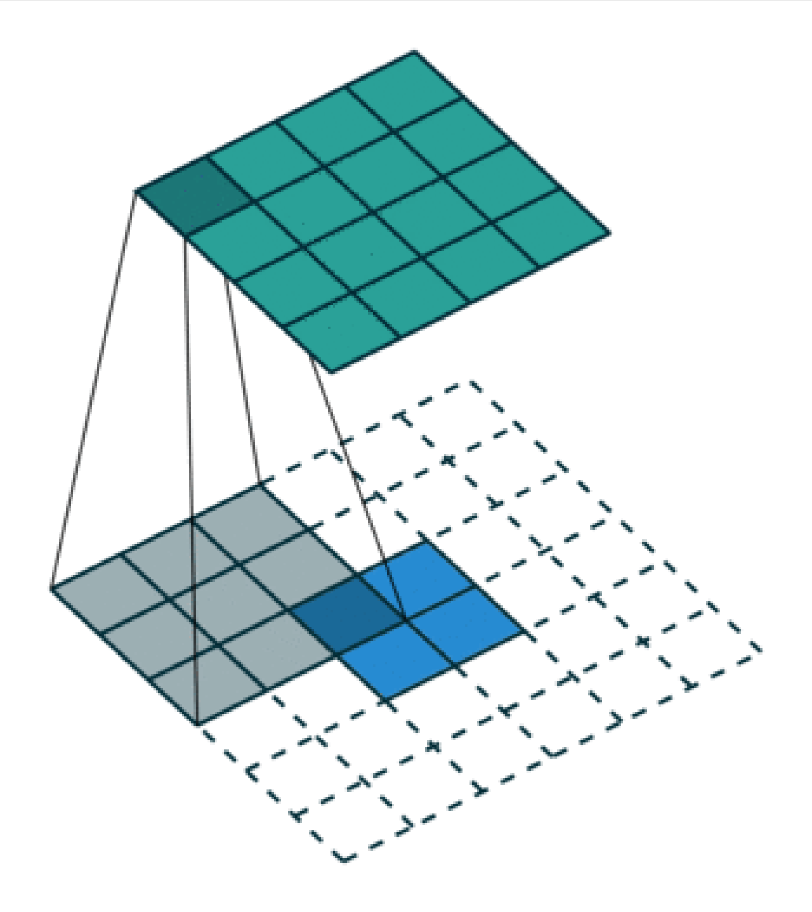
Semantic segmentation



Transposed Convolution



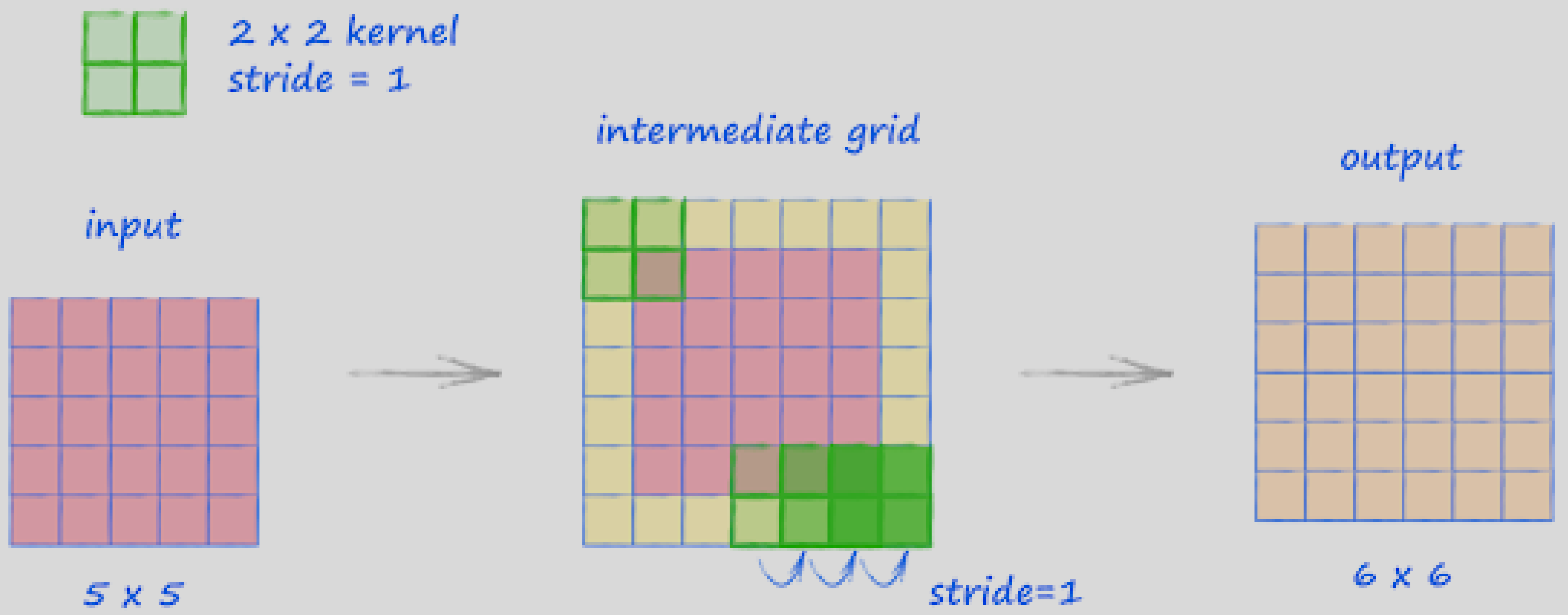
Convolution operation



Transposed convolution operation

Transposed Convolution

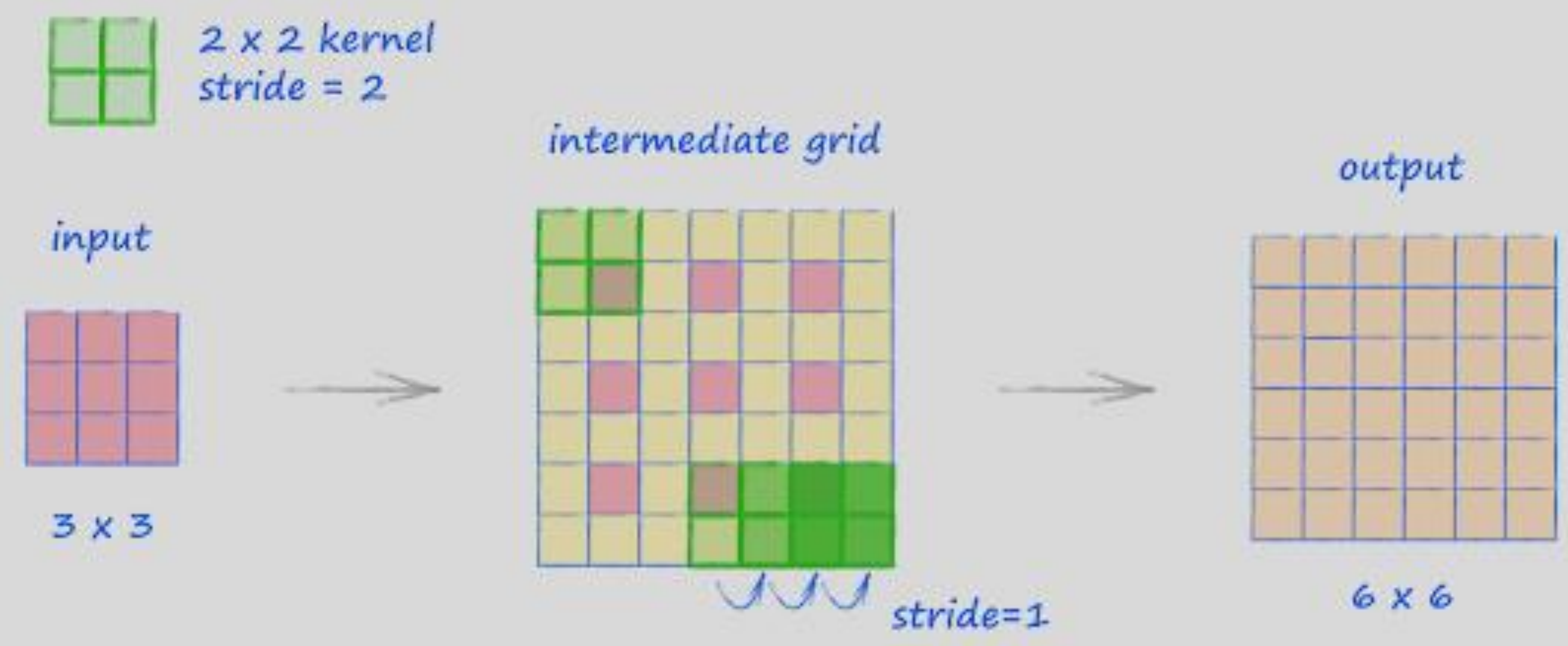
Transposed Convolution with 0 padding, stride 1, 2x2 kernel: Output_size = (input_size-1)*stride – 2*padding + kernel_size + output_padding




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

Transposed Convolution

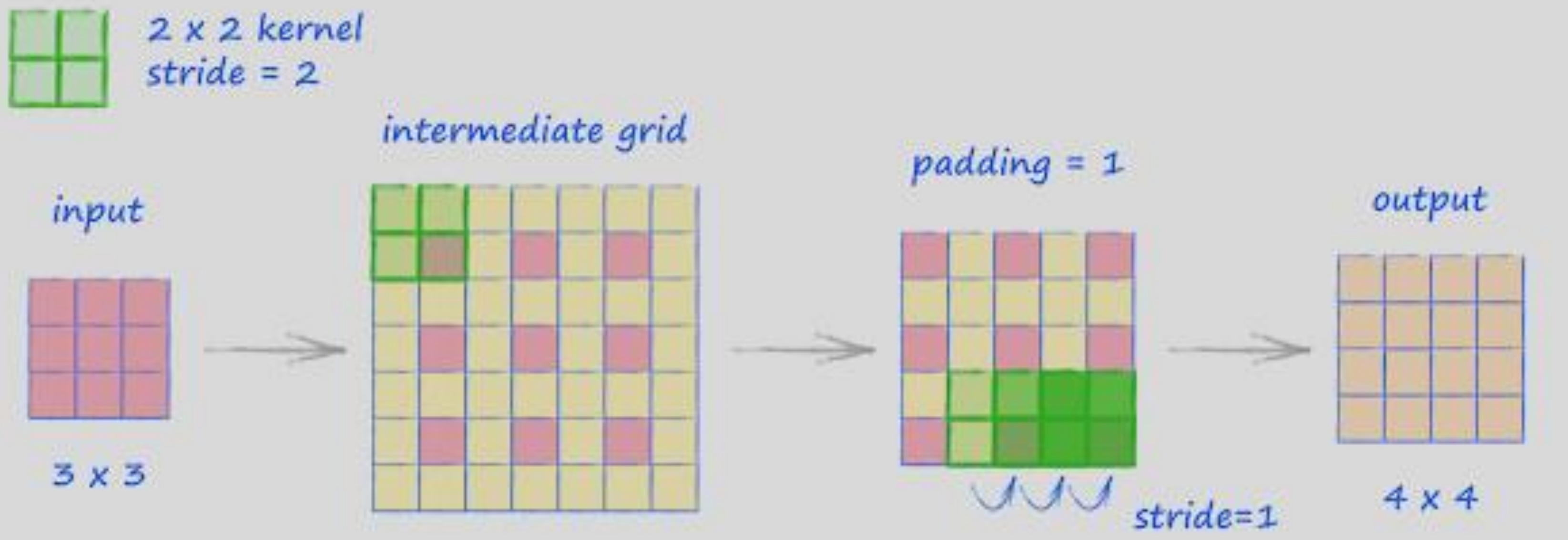
Transposed Convolution with 0 padding, stride 2, 2x2 kernel: Output_size = (input_size-1)*stride – 2*padding + kernel_size + output_padding




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

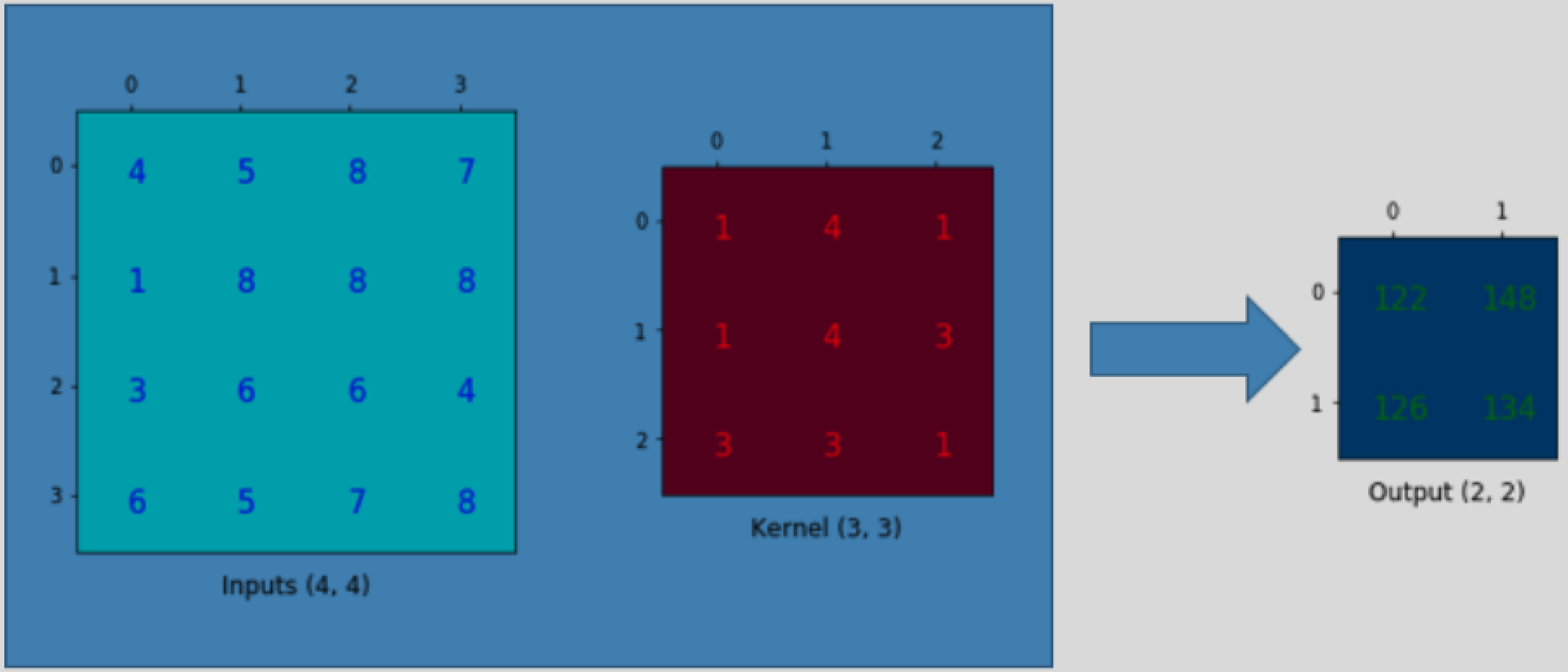
Transposed Convolution

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

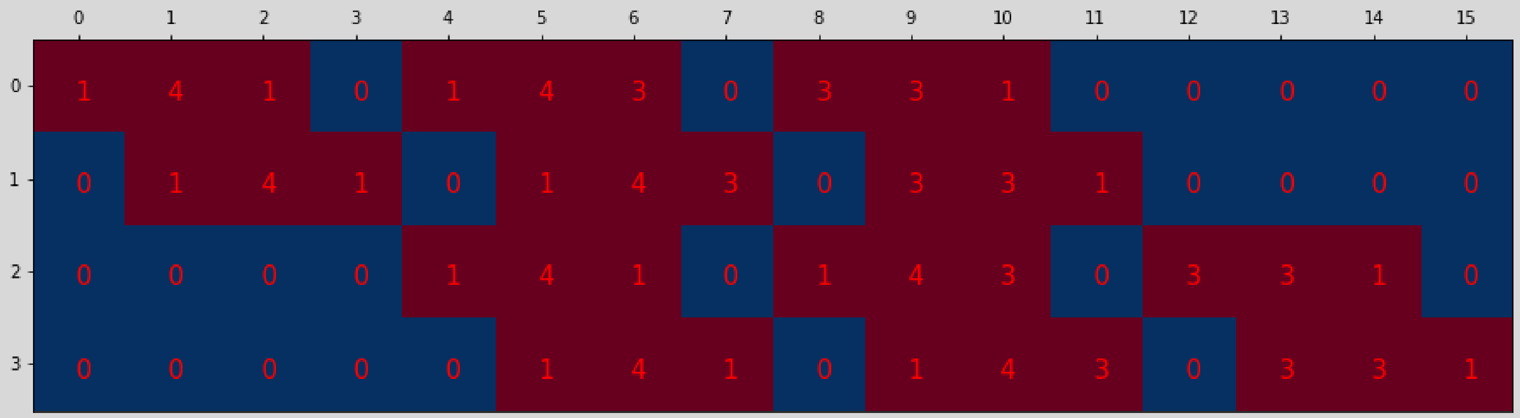



```
nn.ConvTranspose2d(in_channels, out_channels, kernel_size=2, stride=2, padding=1)
```

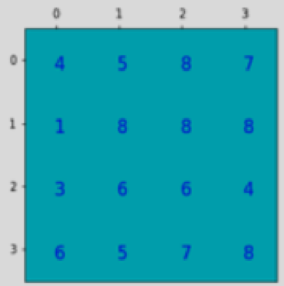
Why called as Transpose Convoluton?



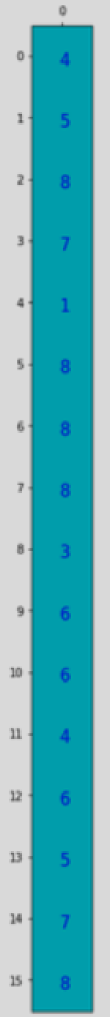
Why called as Transpose Convoluton?



Convolution Matrix (4, 16)

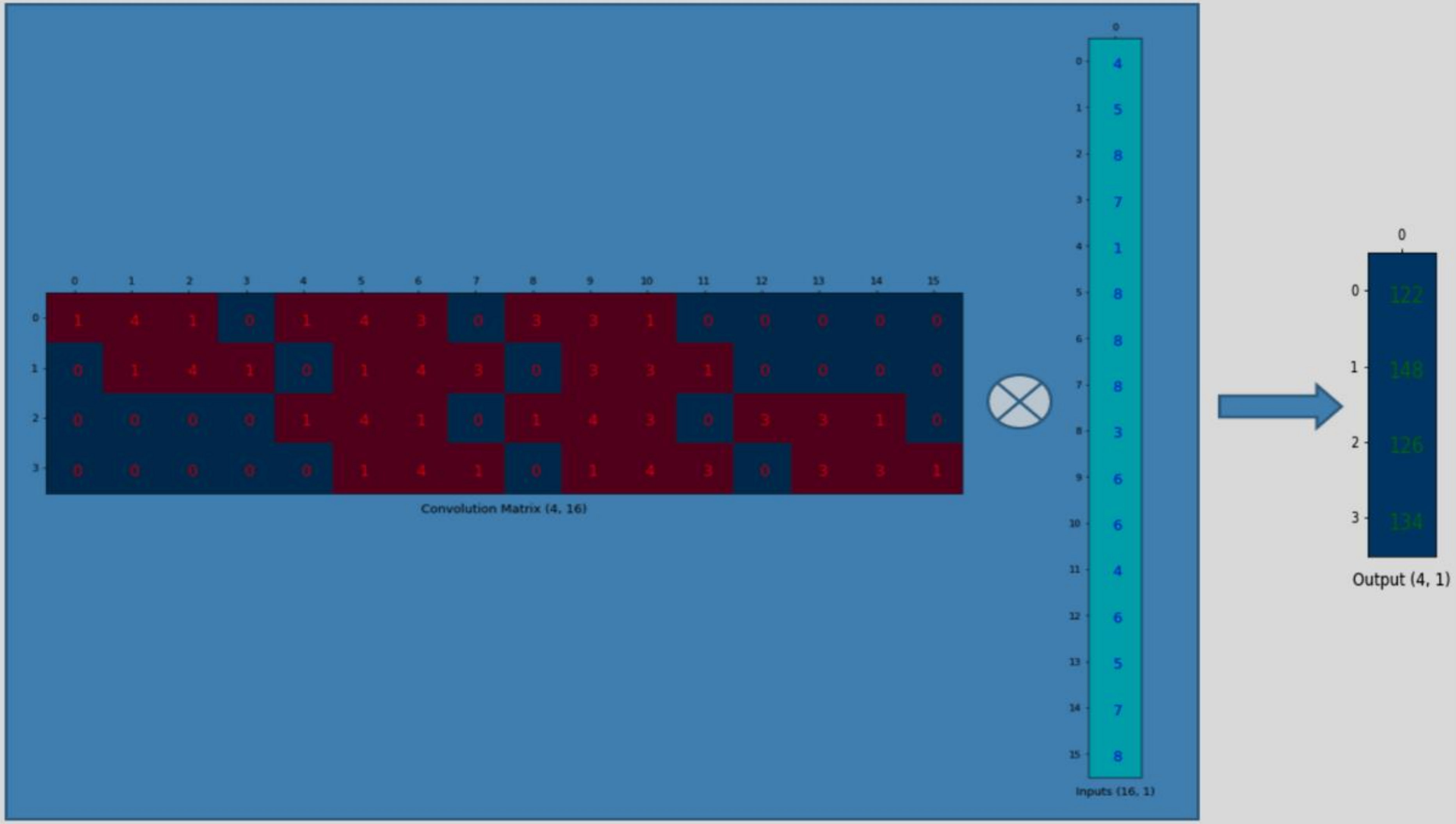


Inputs (4, 4)

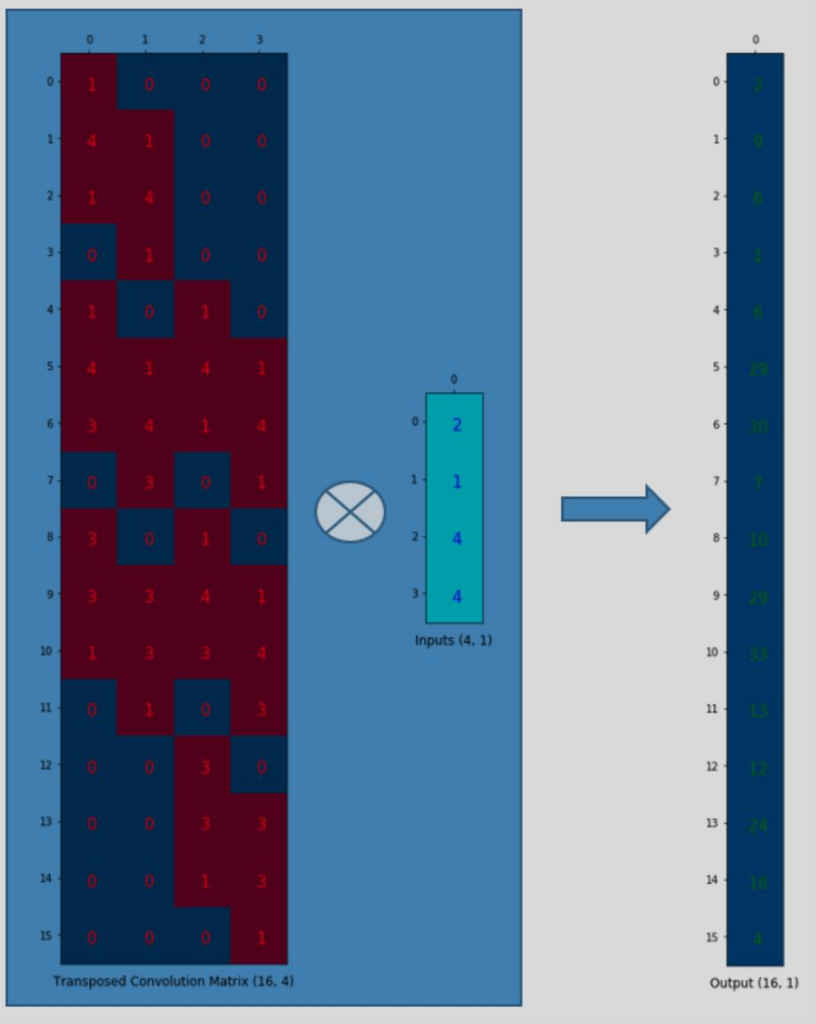


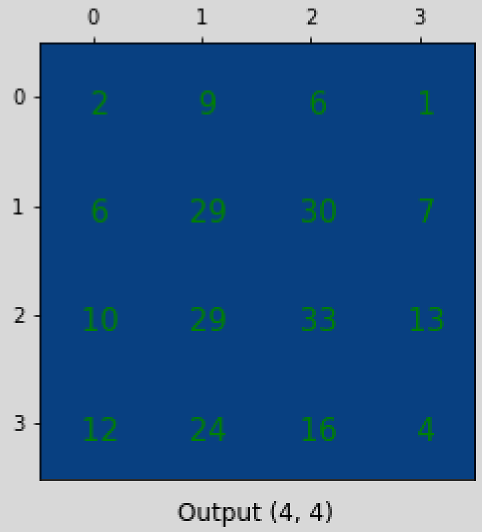
Inputs (16, 1)

Why called as Transpose Convoluton?

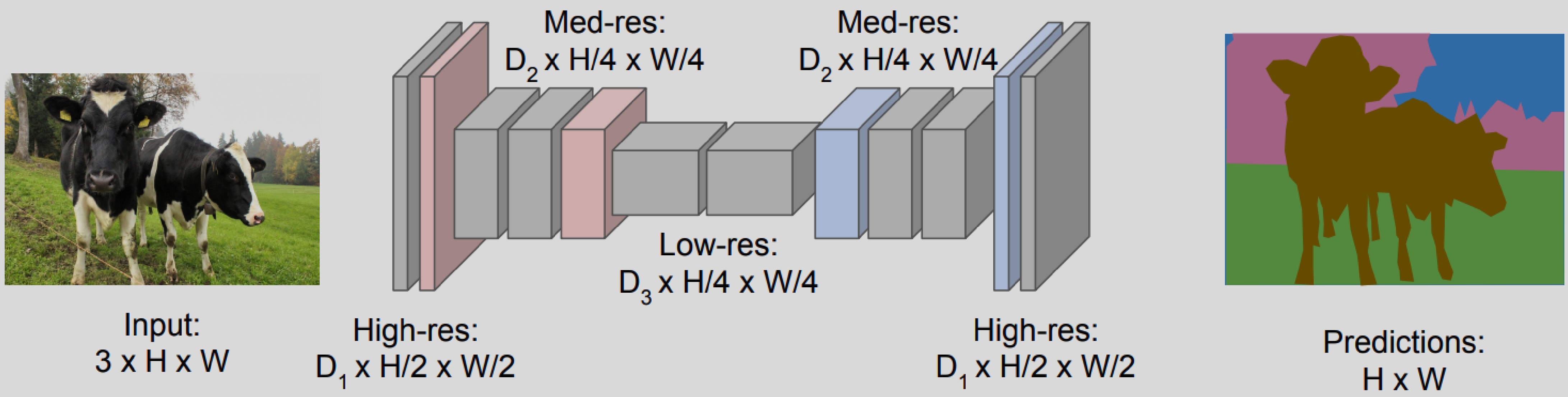


Why called as Transpose Convoluton?





Semantic segmentation



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
```

Encoder-decoder

```
class SegNet(nn.Module):
    def __init__(self, numObj):
        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, numObj, 3, stride=2, padding=1, output_padding=1),
            nn.ReLU()
        )

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

Encoder-decoder

```
numObj = 10
model = SegNet(numObj)
model.train()
criterion = torch.nn.CrossEntropyLoss()

for epoch in range(NUM_EPOCHS):

    for batch in train_dataloader:

        input = torch.autograd.Variable(batch['image'])
        target = torch.autograd.Variable(batch['mask'])

        predicted = model(input)
        output = torch.nn.functional.softmax(predicted, dim=1)

        optimizer.zero_grad()
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()
```

Next class

Remaining Topics:

- Bit more about training CNNs: types of optimizer, large-scale dataset issue.

- Advanced feature for computer vision applications

- (Quiz3, PA3)

- Weakly-/Self-supervised learning, Efficient training via knowledge distillation, continual learning

- Transformer for computer vision applications

- (Final exam)

Thank you!