Segmentations

한양대학교

컴퓨터소프트웨어학부

박현준

Contents

[Segmentation]

FCN : Semantic Segmentation

Mask R-CNN : Instance Segmentation

Panoptic Segmentation

[Video Object Detection]

Deep Feature Flow for Video Recognition

FCN

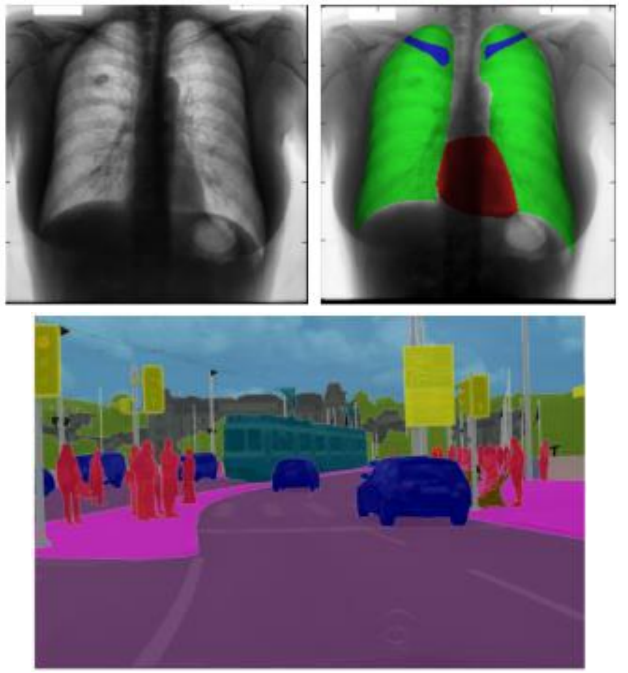
Fully Convolutional Networks for Semantic Segmentation

Introduction

CNN-based Image Classification Network (**AlexNet**, **VGG16**, **GoogLeNet**)

+

Semantic Segmentation



Transfer-Learning

① Visual Recognition
② Detection
③ Instance & Semantic Segmentation

Image Classification → Semantic Segmentation

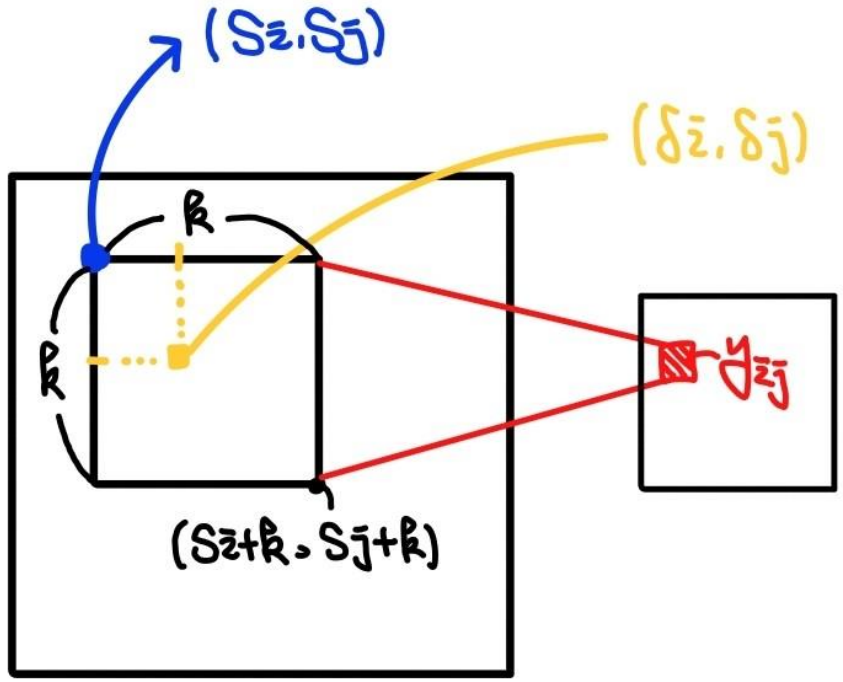
① Convolutionization
② Deconvolution (Upsampling)
③ Skip Architecture

Fully Convolutional Network

ConvNets – Translation Invariant

↳ Operates on local input regions, and depend only on relative spatial coordinates

x_{ij} : Data vector at location (i, j) in a particular layer
y_{ij} : Following Layer
k : kernel size
s : stride or subsampling factor
f_{ks} : layer type (e.g. matmul for conv or avg pool)



$$y_{ij} = f_{ks}\left(\{x_{si+\delta i,\, sj+\delta j}\}_{0 \le \delta i,\, \delta j \le k}\right)$$

- Compute Kernel Size, Stride

$$f_{ks} \circ g_{k's'} = (f \circ g)_{k'+(k-1)s',\, ss'}$$

- Loss Function

$$\ell(x; \theta) = \sum_{ij} \ell'(x_{ij}; \theta)$$

Fully Convolutional Network

Adapting classifiers for dense prediction

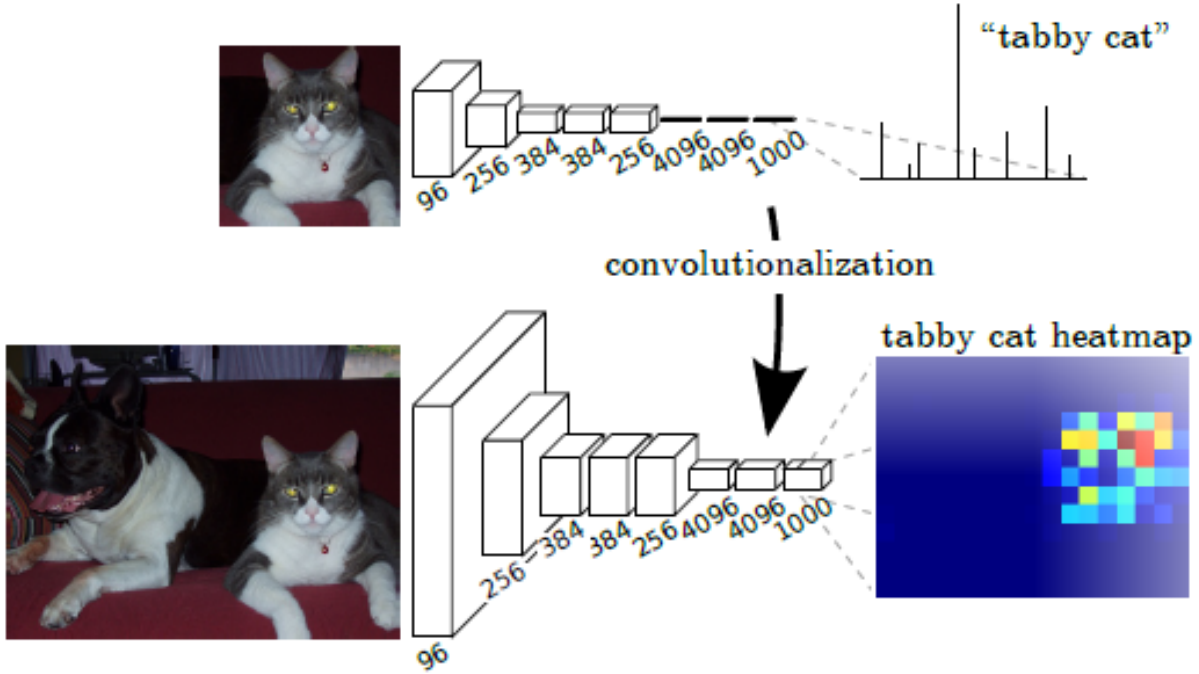
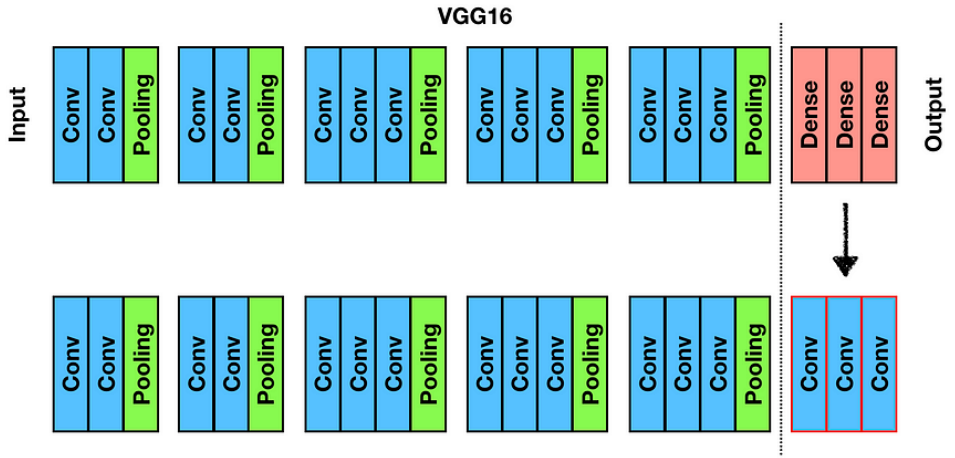


Figure 2. Transforming fully connected layers into convolution layers enables a classification net to output a heatmap. Adding layers and a spatial loss (as in Figure 1) produces an efficient machine for end-to-end dense learning.

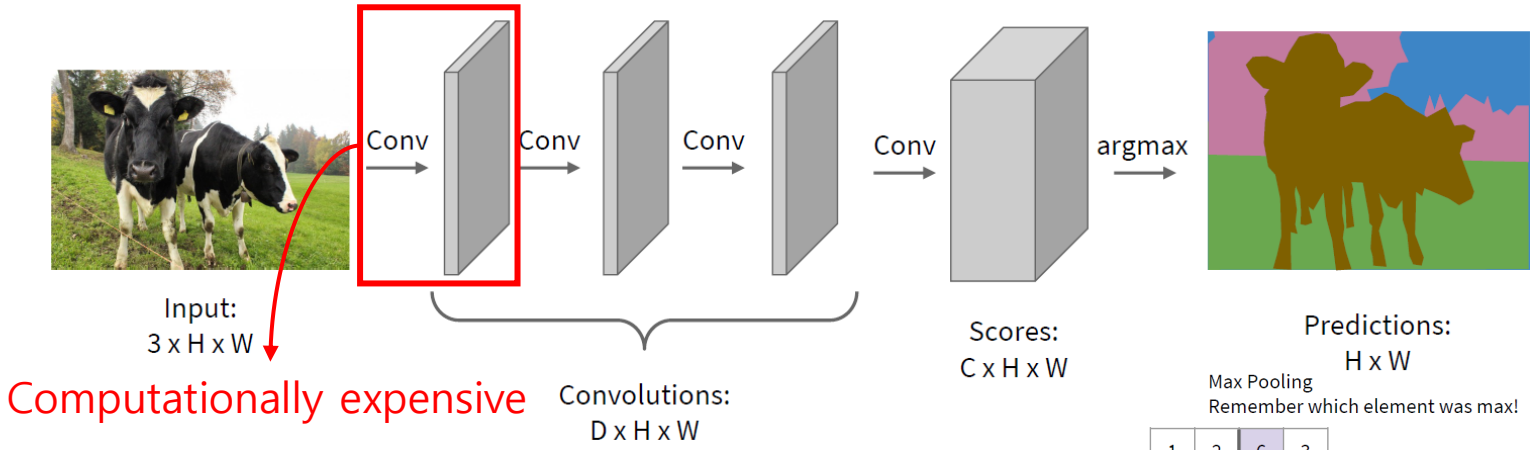


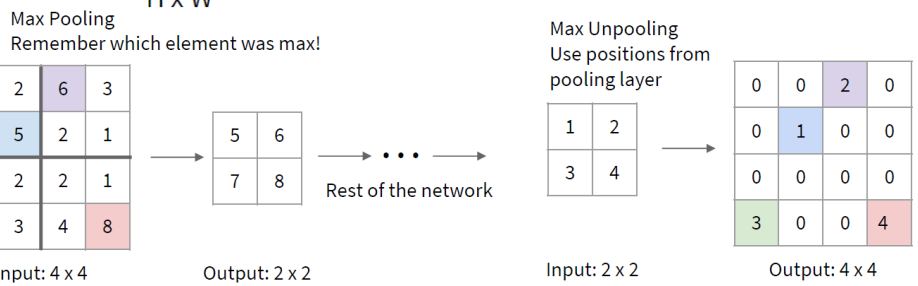
"**Convolutionize**" fc layers

⇒ Preserve spatial information

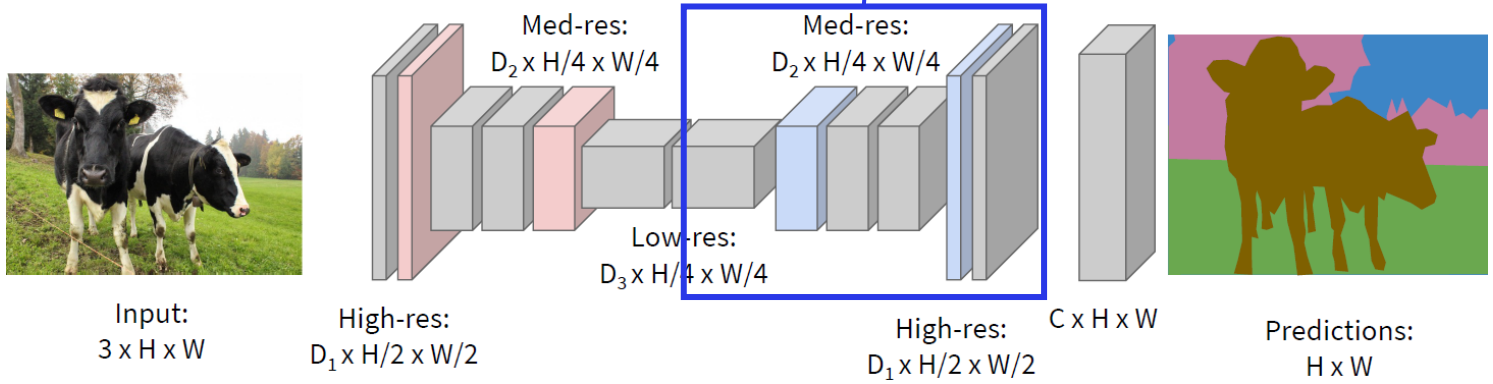
Fully Convolutional Network

Deconvolution (Upsampling)





Max Unpooling



Fully Convolutional Network

Deconvolution (Upsampling)

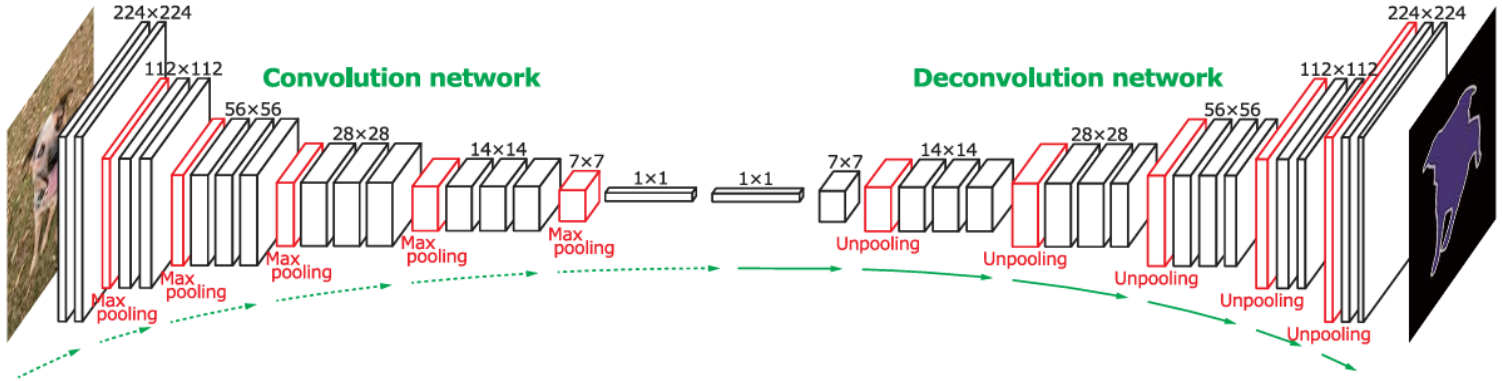


Figure 2. Overall architecture of the proposed network. On top of the convolution network based on VGG 16-layer net, we put a multi-layer deconvolution network to generate the accurate segmentation map of an input proposal. Given a feature representation obtained from the convolution network, dense pixel-wise class prediction map is constructed through multiple series of unpooling, deconvolution and rectification operations.

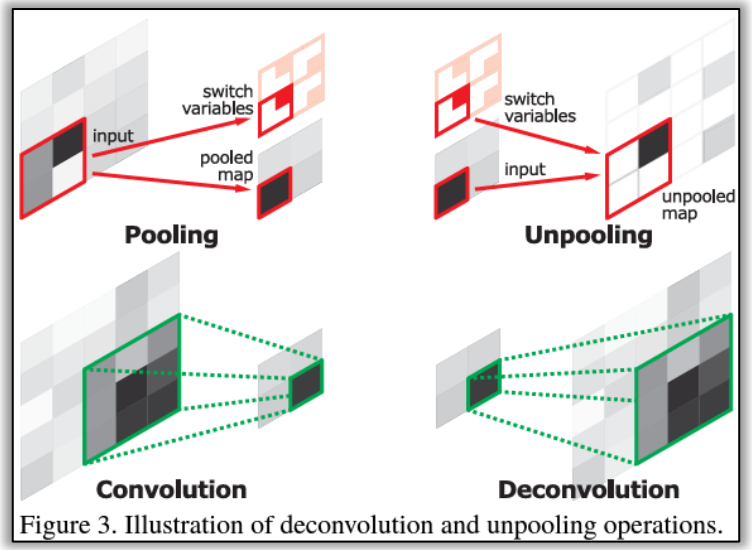
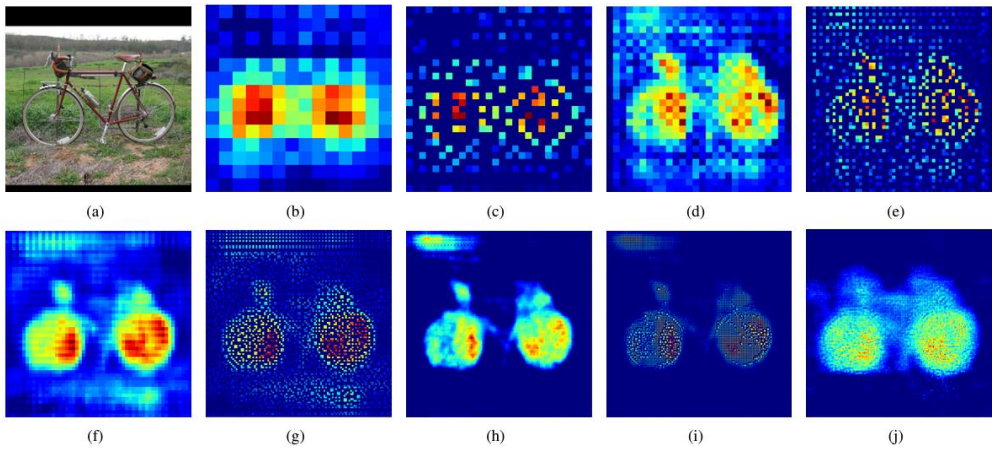
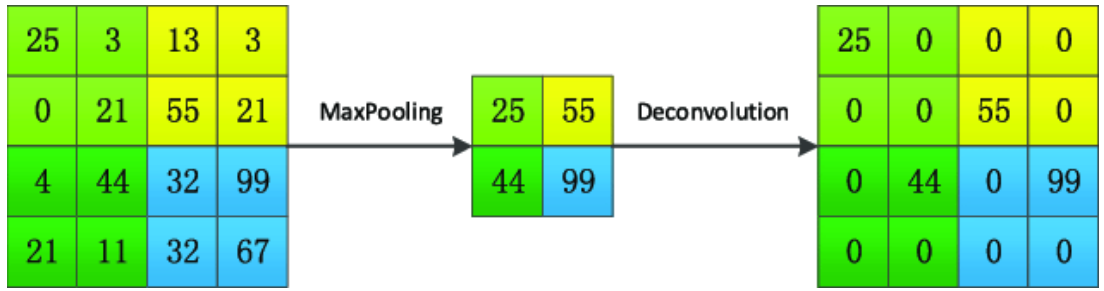


Figure 3. Illustration of deconvolution and unpooling operations.



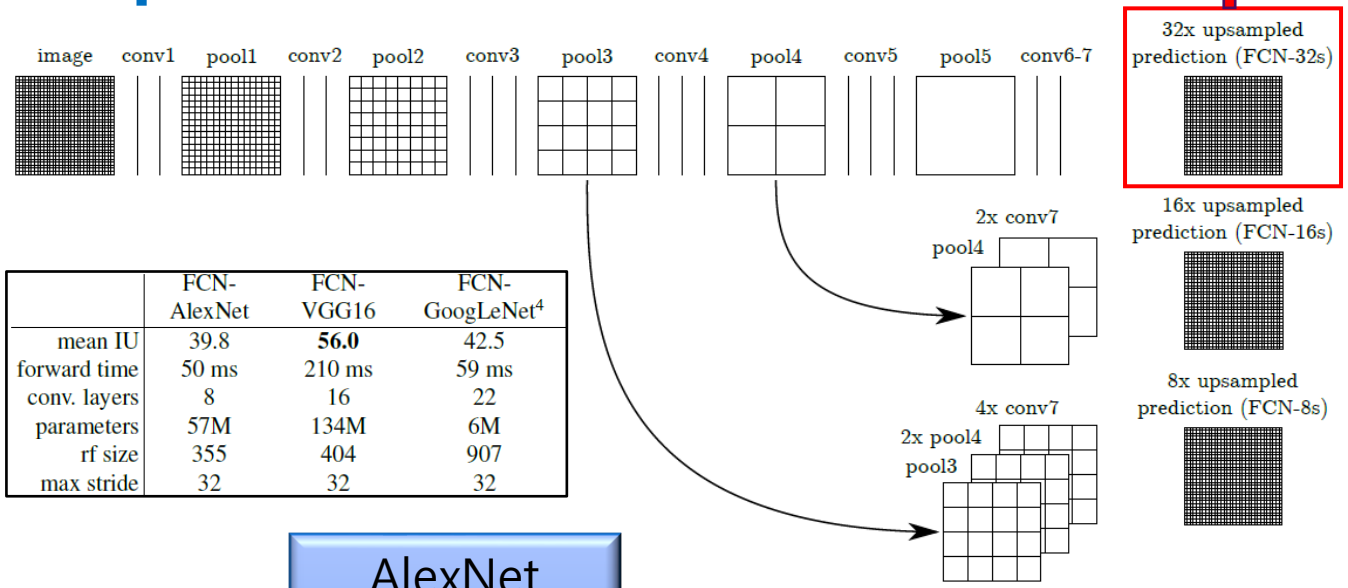


Hyeonwoo Noh, Seunghoon Hong, Bohyung Han. 「***Learning Deconvolution Network for Semantic Segmentation***」*, 2015*

Fully Convolutional Network

Skip Architecture

Limits the scale of detail in the upsampled output



	FCN-AlexNet	FCN-VGG16	FCN-GoogLeNet[4]
mean IU	39.8	**56.0**	42.5
forward time	50 ms	210 ms	59 ms
conv. layers	8	16	22
parameters	57M	134M	6M
rf size	355	404	907
max stride	32	32	32

AlexNet
VGG16
GoogLeNet

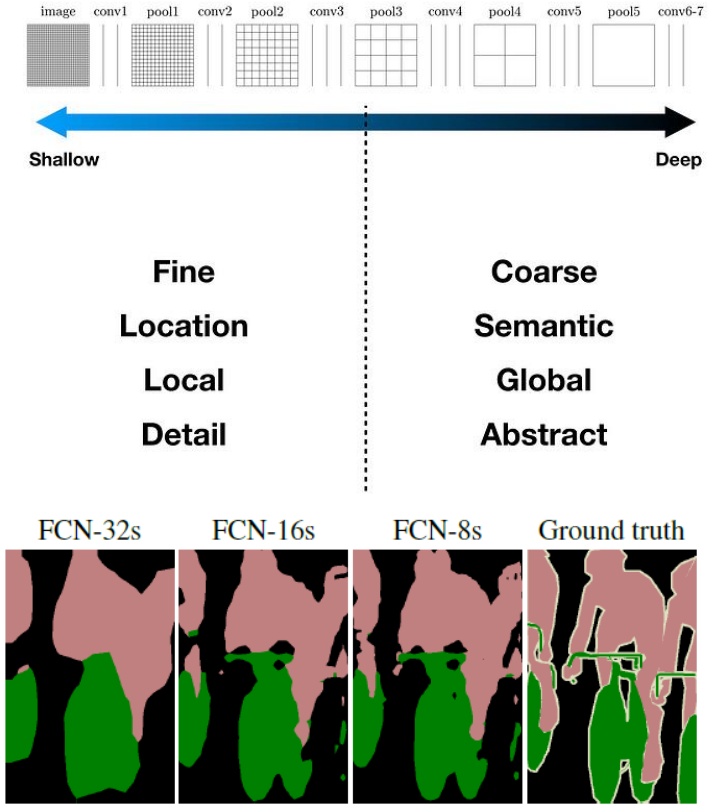
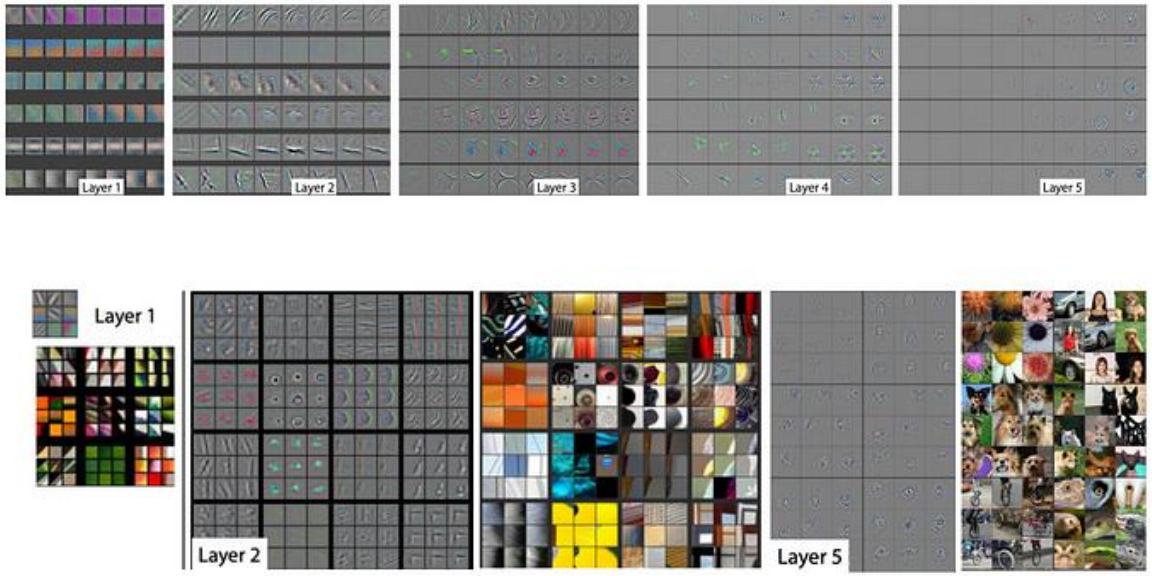


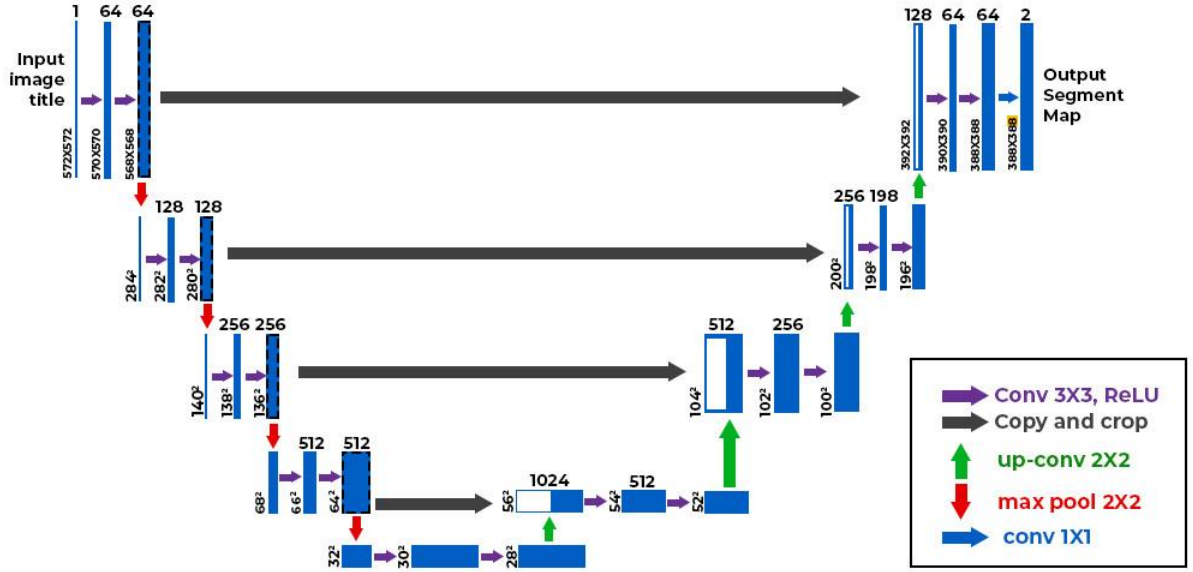
Figure 4. Refining fully convolutional nets by fusing information from layers with different strides improves segmentation detail. The first three images show the output from our 32, 16, and 8 pixel stride nets (see Figure 3).

- ILSVRC classifier + FCNs
- Fine-tuning for segmentation
- Skip connection between layers to fuse coarse, semantic and local, appearance info → learned end-to-end to refine the semantics and spatial precision of the output

Fully Convolutional Network

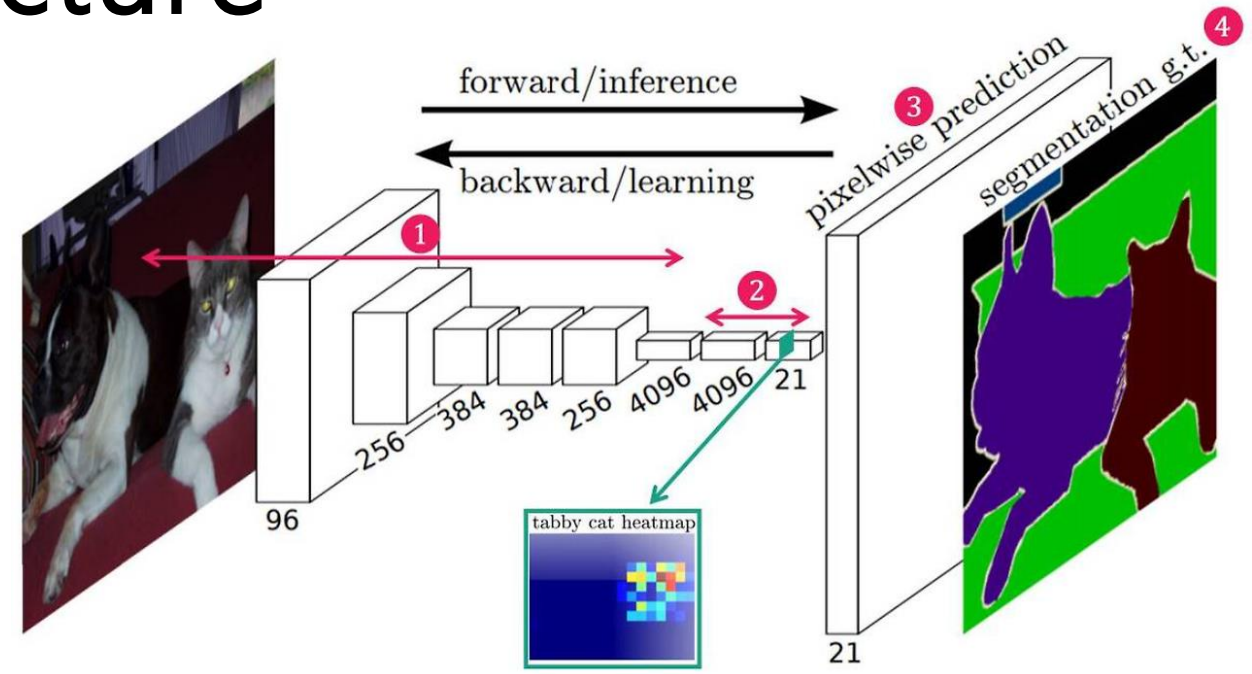
Skip Architecture





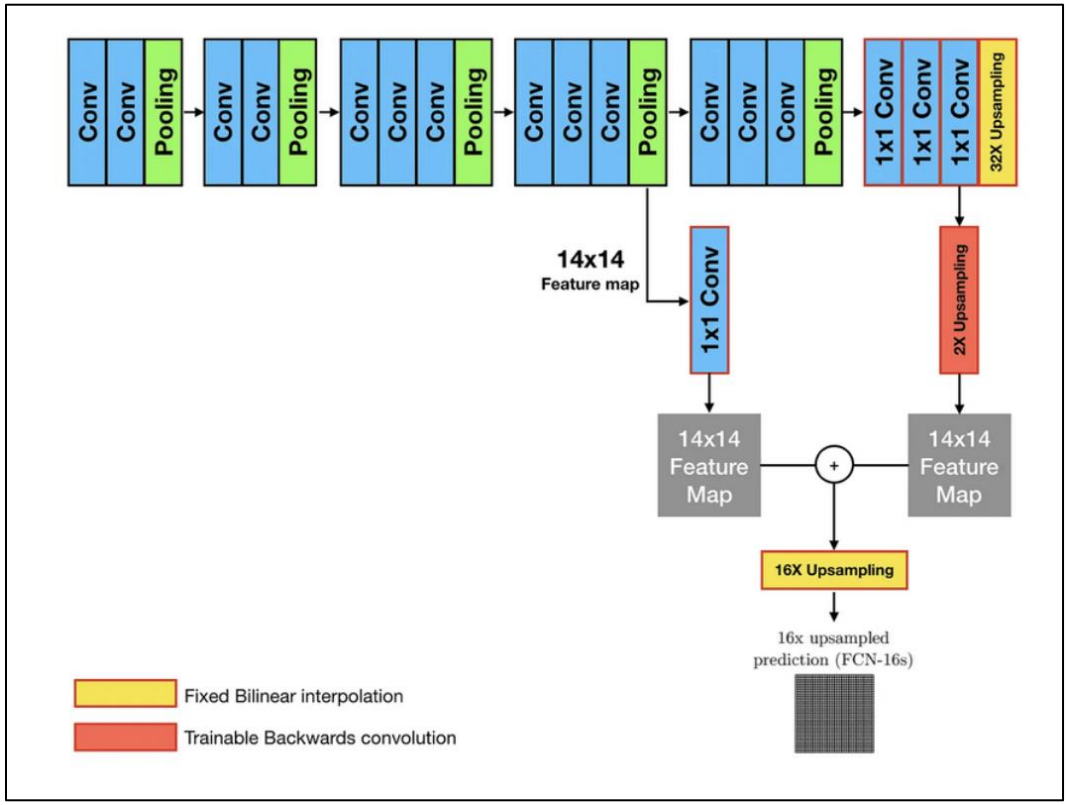
*Matthew D.Zeiler, Rob Furgus.「**Visualizing and Understanding Convolutional Networks**」, 2013*

Architecture

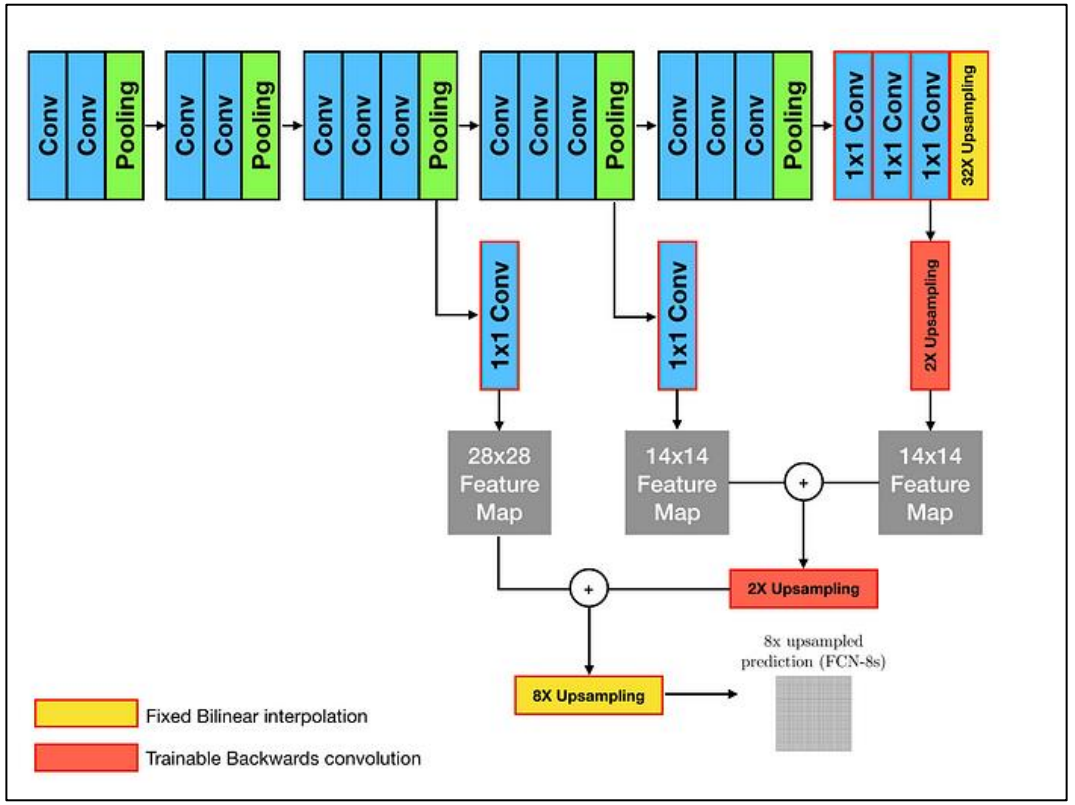


1. Feature map extraction by conv layers
2. Make feature map $\#channel = \#object$ in dataset
   by using 1x1 conv layer (extract class presence heat map)
3. Up-sampling (Transposed convolution)
   → Create map which size is same as input image
4. Training network using difference between final feature map and labeled feature map

Architecture



FCN-16s



FCN-8s

Result

Metrics

- Pixel Accuracy : $\frac{\sum_i n_{ii}}{\sum_i t_i}$
- Mean Accuracy : $\frac{1}{n_{cl}} * \frac{\sum_i n_{ii}}{\sum_i t_i}$
- Mean IoU : $\frac{1}{n_{cl}} * \frac{\sum_i n_{ii}}{t_i + \sum_j n_{ji} - n_{ii}}$
- Frequency weighted IoU : $(\sum_k t_k)^{-1} \frac{\sum_i t_i n_{ii}}{t_i + \sum_j n_{ji} - n_{ii}}$

n_{ij} : #pixels ∈ $class_i$ predicted to belong to $class_j$

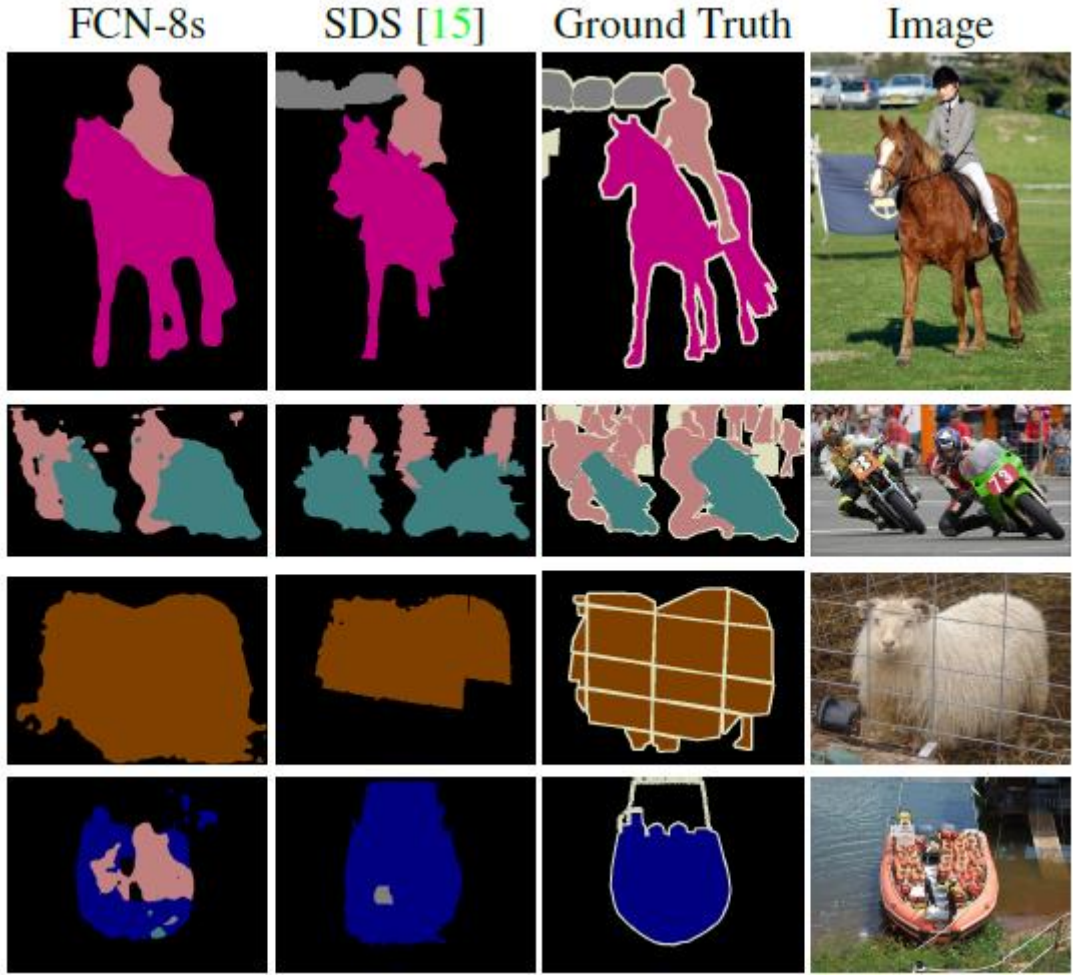
n_{cl} : #class

$t_i = \sum_j n_{ij}$: #pixels ∈ $class_i$

Result

	pixel acc.	mean acc.	mean IU	f.w. IU
Gupta *et al.* [13]	60.3	-	28.6	47.0
FCN-32s RGB	60.0	42.2	29.2	43.9
FCN-32s RGBD	61.5	42.4	30.5	45.5
FCN-32s HHA	57.1	35.2	24.2	40.4
FCN-32s RGB-HHA	64.3	44.9	32.8	48.0
FCN-16s RGB-HHA	**65.4**	**46.1**	**34.0**	**49.5**

	pixel acc.	mean acc.	mean IU	f.w. IU	geom. acc.
Liu *et al.* [23]	76.7	-	-	-	-
Tighe *et al.* [33]	-	-	-	-	90.8
Tighe *et al.* [34] 1	75.6	41.1	-	-	-
Tighe *et al.* [34] 2	78.6	39.2	-	-	-
Farabet *et al.* [7] 1	72.3	50.8	-	-	-
Farabet *et al.* [7] 2	78.5	29.6	-	-	-
Pinheiro *et al.* [28]	77.7	29.8	-	-	-
FCN-16s	**85.2**	**51.7**	39.5	76.1	**94.3**



Conclusion

FCN

Strength

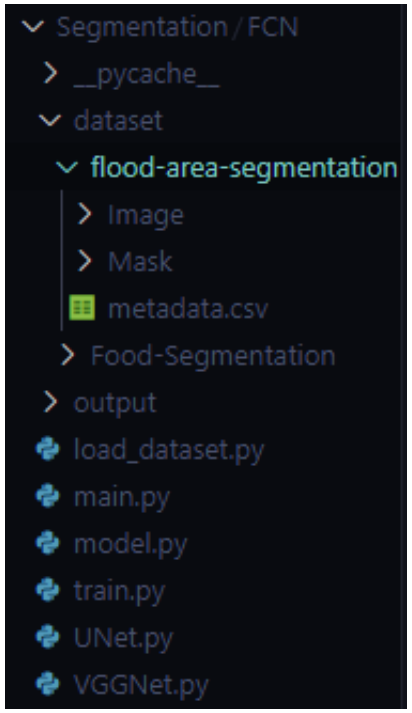
- "Skip architecture" ⇒ Performance ↑ dramatically
- Apply image classification network to image segmentation task
- Convolutionize fc layer

(∵ lose spatial info, fixed input image size, too much #parameters)

Implementation

<Dataset>
: Flood Area Segmentation (from Kaggle)

- Images : 290
- Labels : 290




```
15:46:03  jaehak(HJo0on)  ...Hyunjoon/Segmentation/FCN
$ python model.py
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 64, 224, 224]           1,792
              ReLU-2         [-1, 64, 224, 224]               0
            Conv2d-3         [-1, 64, 224, 224]          36,928
              ReLU-4         [-1, 64, 224, 224]               0
         MaxPool2d-5         [-1, 64, 112, 112]               0
            Conv2d-6        [-1, 128, 112, 112]          73,856
              ReLU-7        [-1, 128, 112, 112]               0
            Conv2d-8        [-1, 128, 112, 112]         147,584
              ReLU-9        [-1, 128, 112, 112]               0
        MaxPool2d-10          [-1, 128, 56, 56]               0
           Conv2d-11          [-1, 256, 56, 56]         295,168
             ReLU-12          [-1, 256, 56, 56]               0
           Conv2d-13          [-1, 256, 56, 56]         590,080
             ReLU-14          [-1, 256, 56, 56]               0
        MaxPool2d-15          [-1, 256, 28, 28]               0
           Conv2d-16          [-1, 512, 28, 28]       1,180,160
             ReLU-17          [-1, 512, 28, 28]               0
           Conv2d-18          [-1, 512, 28, 28]       2,359,808
             ReLU-19          [-1, 512, 28, 28]               0
        MaxPool2d-20          [-1, 512, 14, 14]               0
           Conv2d-21          [-1, 512, 14, 14]       2,359,808
             ReLU-22          [-1, 512, 14, 14]               0
           Conv2d-23          [-1, 512, 14, 14]       2,359,808
             ReLU-24          [-1, 512, 14, 14]               0
        MaxPool2d-25            [-1, 512, 7, 7]               0
  ConvTranspose2d-26          [-1, 512, 14, 14]       2,359,808
             ReLU-27          [-1, 512, 14, 14]               0
      BatchNorm2d-28          [-1, 512, 14, 14]           1,024
  ConvTranspose2d-29          [-1, 256, 28, 28]       1,179,904
             ReLU-30          [-1, 256, 28, 28]               0
      BatchNorm2d-31          [-1, 256, 28, 28]             512
  ConvTranspose2d-32          [-1, 128, 56, 56]         295,040
             ReLU-33          [-1, 128, 56, 56]               0
      BatchNorm2d-34          [-1, 128, 56, 56]             256
  ConvTranspose2d-35         [-1, 64, 112, 112]          73,792
             ReLU-36         [-1, 64, 112, 112]               0
      BatchNorm2d-37         [-1, 64, 112, 112]             128
  ConvTranspose2d-38         [-1, 32, 224, 224]          18,464
             ReLU-39         [-1, 32, 224, 224]               0
      BatchNorm2d-40         [-1, 32, 224, 224]              64
           Conv2d-41          [-1, 2, 224, 224]              66
================================================================
Total params: 13,334,050
Trainable params: 13,334,050
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.57
Forward/backward pass size (MB): 270.46
```

dataset > flood-area-segmentation > metadata.csv > data

```
Image,Mask
0.jpg,0.png
1.jpg,1.png
2.jpg,2.png
3.jpg,3.png
4.jpg,4.png
5.jpg,5.png
6.jpg,6.png
7.jpg,7.png
8.jpg,8.png
9.jpg,9.png
10.jpg,10.png
11.jpg,11.png
12.jpg,12.png
13.jpg,13.png
14.jpg,14.png
15.jpg,15.png
16.jpg,16.png
17.jpg,17.png
18.jpg,18.png
19.jpg,19.png
20.jpg,20.png
```

Implementation

FCN

```
import torch
import torch.nn as nn
from torchsummary import summary as model_summary

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

        self.block1 = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        self.block2 = nn.Sequential(
            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(128, 128, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        self.block3 = nn.Sequential(
            nn.Conv2d(128, 256, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        self.block4 = nn.Sequential(
            nn.Conv2d(256, 512, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        self.block5 = nn.Sequential(
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.Conv2d(512, 512, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )

        self.relu    = nn.ReLU(inplace=True)
        self.deconv1 = nn.ConvTranspose2d(512, 512, kernel_size=3, stride=2, padding=1, dilation=1, output_padding=1)
        self.bn1     = nn.BatchNorm2d(512)
        self.deconv2 = nn.ConvTranspose2d(512, 256, kernel_size=3, stride=2, padding=1, dilation=1, output_padding=1)
        self.bn2     = nn.BatchNorm2d(256)
        self.deconv3 = nn.ConvTranspose2d(256, 128, kernel_size=3, stride=2, padding=1, dilation=1, output_padding=1)
        self.bn3     = nn.BatchNorm2d(128)
        self.deconv4 = nn.ConvTranspose2d(128, 64, kernel_size=3, stride=2, padding=1, dilation=1, output_padding=1)
        self.bn4     = nn.BatchNorm2d(64)
        self.deconv5 = nn.ConvTranspose2d(64, 32, kernel_size=3, stride=2, padding=1, dilation=1, output_padding=1)
        self.bn5     = nn.BatchNorm2d(32)
        self.classifier = nn.Conv2d(32, 2, kernel_size=1)

    def forward(self, x):
        x = self.block1(x)
        x1 = x
        x = self.block2(x)
        x2 = x
        x = self.block3(x)
        x3 = x
        x = self.block4(x)
        x4 = x
        x = self.block5(x)
        x5 = x

        score = self.bn1(self.relu(self.deconv1(x5)))
        score = score + x4
        score = self.bn2(self.relu(self.deconv2(score)))
        score = score + x3
        score = self.bn3(self.relu(self.deconv3(score)))
        score = score + x2
        score = self.bn4(self.relu(self.deconv4(score)))
        score = score + x1
        score = self.bn5(self.relu(self.deconv5(score)))
        score = self.classifier(score)

        return score
```

Pre-trained VGG16

```
import torch
import torch.nn as nn
from torchvision import models
from torchsummary import summary as model_summary

ranges = {'vgg16': ((0, 5), (5, 10), (10, 17), (17, 24), (24, 31))}

class VGGNet(nn.Module):
    def __init__(self, pretrained=True):
        super(VGGNet, self).__init__()
        self.ranges = ranges['vgg16']
        self.features = models.vgg16(weights=pretrained).features

    def forward(self, x):
        output = {}
        for idx in range(len(self.ranges)):
            for layer in range(self.ranges[idx][0], self.ranges[idx][1]):
                x = self.features[layer](x)
            output["x%d"%(idx+1)] = x
        return output


class FCNs(nn.Module):
    def __init__(self, pretrained_net, n_class):
        super().__init__()
        self.n_class = n_class
        self.pretrained_net = pretrained_net
        self.relu    = nn.ReLU(inplace=True)
        self.deconv1 = nn.ConvTranspose2d(512, 512, kernel_size=3, stride=2, padding=1, dilation=1, output_padding=1)
        self.bn1     = nn.BatchNorm2d(512)
        self.deconv2 = nn.ConvTranspose2d(512, 256, kernel_size=3, stride=2, padding=1, dilation=1, output_padding=1)
        self.bn2     = nn.BatchNorm2d(256)
        self.deconv3 = nn.ConvTranspose2d(256, 128, kernel_size=3, stride=2, padding=1, dilation=1, output_padding=1)
        self.bn3     = nn.BatchNorm2d(128)
        self.deconv4 = nn.ConvTranspose2d(128, 64, kernel_size=3, stride=2, padding=1, dilation=1, output_padding=1)
        self.bn4     = nn.BatchNorm2d(64)
        self.deconv5 = nn.ConvTranspose2d(64, 32, kernel_size=3, stride=2, padding=1, dilation=1, output_padding=1)
        self.bn5     = nn.BatchNorm2d(32)
        self.classifier = nn.Conv2d(32, n_class, kernel_size=1)

    def forward(self, x):
        output = self.pretrained_net(x)

        x5 = output['x5']
        x4 = output['x4']
        x3 = output['x3']
        x2 = output['x2']
        x1 = output['x1']

        score = self.bn1(self.relu(self.deconv1(x5)))
        score = score + x4
        score = self.bn2(self.relu(self.deconv2(score)))
        score = score + x3
        score = self.bn3(self.relu(self.deconv3(score)))
        score = score + x2
        score = self.bn4(self.relu(self.deconv4(score)))
        score = score + x1
        score = self.bn5(self.relu(self.deconv5(score)))
        score = self.classifier(score)

        return score

vgg16 = VGGNet(pretrained=True)
model = FCNs(vgg16, 2)
model
```

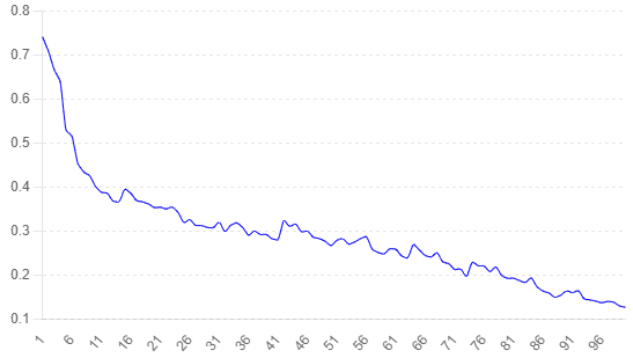
UNet

```
import torch
import torch.nn as nn

def dual_conv(in_channel, out_channel):
    conv = nn.Sequential(
        nn.Conv2d(in_channel, out_channel, kernel_size=3, padding=1),
        nn.ReLU(inplace=True),
        nn.Conv2d(out_channel, out_channel, kernel_size=3, padding=1),
        nn.ReLU(inplace=True),
    )
    return conv

def crop_tensor(target_tensor, tensor):
    target_size = target_tensor.size()[2]
    tensor_size = tensor.size()[2]
    delta = tensor_size - target_size
    delta = delta // 2

    return tensor[:, :, delta:tensor_size - delta, delta:tensor_size - delta]

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

        # Left side (contracting path)
        self.down_conv1 = dual_conv(3, 64)
        self.down_conv2 = dual_conv(64, 128)
        self.down_conv3 = dual_conv(128, 256)
        self.down_conv4 = dual_conv(256, 512)
        self.down_conv5 = dual_conv(512, 1024)
        self.maxpool = nn.MaxPool2d(kernel_size=2, stride=2)

        # Right side (expansion path)
        self.trans1 = nn.ConvTranspose2d(1024, 512, kernel_size=2, stride=2)
        self.up_conv1 = dual_conv(1024, 512)
        self.trans2 = nn.ConvTranspose2d(512, 256, kernel_size=2, stride=2)
        self.up_conv2 = dual_conv(512, 256)
        self.trans3 = nn.ConvTranspose2d(256, 128, kernel_size=2, stride=2)
        self.up_conv3 = dual_conv(256, 128)
        self.trans4 = nn.ConvTranspose2d(128, 64, kernel_size=2, stride=2)
        self.up_conv4 = dual_conv(128, 64)

        # output layer
        self.out = nn.Conv2d(64, 2, kernel_size=1)

    def forward(self, image):
        # Left side
        x1 = self.down_conv1(image)
        x2 = self.maxpool(x1)
        x3 = self.down_conv2(x2)
        x4 = self.maxpool(x3)
        x5 = self.down_conv3(x4)
        x6 = self.maxpool(x5)
        x7 = self.down_conv4(x6)
        x8 = self.maxpool(x7)
        x9 = self.down_conv5(x8)

        # Right side
        x = self.trans1(x9)
        y = crop_tensor(x, x7)
        x = self.up_conv1(torch.cat([x, y], 1))

        x = self.trans2(x)
        y = crop_tensor(x, x5)
        x = self.up_conv2(torch.cat([x, y], 1))

        x = self.trans3(x)
        y = crop_tensor(x, x3)
        x = self.up_conv3(torch.cat([x, y], 1))

        x = self.trans4(x)
        y = crop_tensor(x, x1)
        x = self.up_conv4(torch.cat([x, y], 1))

        x = self.out(x)

        return x
```

Implementation

FCN

Epoch=25	Epoch=100		
Epoch 1/25, Loss: 0.6961	Epoch 1/100, Loss: 0.7409	Epoch 35/100, Loss: 0.3079	Epoch 69/100, Loss: 0.2303
Epoch 2/25, Loss: 0.6707	Epoch 2/100, Loss: 0.7003	Epoch 36/100, Loss: 0.2912	Epoch 70/100, Loss: 0.2258
Epoch 3/25, Loss: 0.6345	Epoch 3/100, Loss: 0.6663	Epoch 37/100, Loss: 0.2999	Epoch 71/100, Loss: 0.2193
Epoch 4/25, Loss: 0.5762	Epoch 4/100, Loss: 0.6038	Epoch 38/100, Loss: 0.2923	Epoch 72/100, Loss: 0.2154
Epoch 5/25, Loss: 0.5114	Epoch 5/100, Loss: 0.5300	Epoch 39/100, Loss: 0.2920	Epoch 73/100, Loss: 0.2402
Epoch 6/25, Loss: 0.4536	Epoch 6/100, Loss: 0.5136	Epoch 40/100, Loss: 0.2825	Epoch 74/100, Loss: 0.2254
Epoch 7/25, Loss: 0.4294	Epoch 7/100, Loss: 0.4545	Epoch 41/100, Loss: 0.3412	Epoch 75/100, Loss: 0.2157
Epoch 8/25, Loss: 0.4227	Epoch 8/100, Loss: 0.4434	Epoch 42/100, Loss: 0.3230	Epoch 76/100, Loss: 0.2332
Epoch 9/25, Loss: 0.3976	Epoch 9/100, Loss: 0.4258	Epoch 43/100, Loss: 0.3113	Epoch 77/100, Loss: 0.2241
Epoch 10/25, Loss: 0.3728	Epoch 10/100, Loss: 0.4016	Epoch 44/100, Loss: 0.3154	Epoch 78/100, Loss: 0.2084
Epoch 11/25, Loss: 0.3540	Epoch 11/100, Loss: 0.3888	Epoch 45/100, Loss: 0.2988	Epoch 79/100, Loss: 0.2112
Epoch 12/25, Loss: 0.3486	Epoch 12/100, Loss: 0.3859	Epoch 46/100, Loss: 0.2957	Epoch 80/100, Loss: 0.2189
Epoch 13/25, Loss: 0.3501	Epoch 13/100, Loss: 0.3684	Epoch 47/100, Loss: 0.2863	Epoch 81/100, Loss: 0.1962
Epoch 14/25, Loss: 0.3627	Epoch 14/100, Loss: 0.3579	Epoch 48/100, Loss: 0.2831	Epoch 82/100, Loss: 0.1925
Epoch 15/25, Loss: 0.3440	Epoch 15/100, Loss: 0.3944	Epoch 49/100, Loss: 0.2709	Epoch 83/100, Loss: 0.1835
Epoch 16/25, Loss: 0.3471	Epoch 16/100, Loss: 0.3856	Epoch 50/100, Loss: 0.2675	Epoch 84/100, Loss: 0.1892
Epoch 17/25, Loss: 0.3410	Epoch 17/100, Loss: 0.3694	Epoch 51/100, Loss: 0.2708	Epoch 85/100, Loss: 0.1737
Epoch 18/25, Loss: 0.3186	Epoch 18/100, Loss: 0.3663	Epoch 52/100, Loss: 0.2825	Epoch 86/100, Loss: 0.1638
Epoch 19/25, Loss: 0.3102	Epoch 19/100, Loss: 0.3616	Epoch 53/100, Loss: 0.2710	Epoch 87/100, Loss: 0.1585
Epoch 20/25, Loss: 0.3048	Epoch 20/100, Loss: 0.3534	Epoch 54/100, Loss: 0.2751	Epoch 88/100, Loss: 0.1546
Epoch 21/25, Loss: 0.3012	Epoch 21/100, Loss: 0.3412	Epoch 55/100, Loss: 0.2820	Epoch 89/100, Loss: 0.1629
Epoch 22/25, Loss: 0.2929	Epoch 22/100, Loss: 0.3504	Epoch 56/100, Loss: 0.2686	Epoch 90/100, Loss: 0.1623
Epoch 23/25, Loss: 0.2925	Epoch 23/100, Loss: 0.3543	Epoch 57/100, Loss: 0.2594	Epoch 91/100, Loss: 0.1644
Epoch 24/25, Loss: 0.3079	Epoch 24/100, Loss: 0.3339	Epoch 58/100, Loss: 0.2515	Epoch 92/100, Loss: 0.1646
Epoch 25/25, Loss: 0.2950	Epoch 25/100, Loss: 0.3198	Epoch 59/100, Loss: 0.2487	Epoch 93/100, Loss: 0.1638
Training complete.	Epoch 26/100, Loss: 0.3257	Epoch 60/100, Loss: 0.2594	Epoch 94/100, Loss: 0.1538
	Epoch 27/100, Loss: 0.3103	Epoch 61/100, Loss: 0.2585	Epoch 95/100, Loss: 0.1462
	Epoch 28/100, Loss: 0.3126	Epoch 62/100, Loss: 0.2438	Epoch 96/100, Loss: 0.1394
	Epoch 29/100, Loss: 0.3082	Epoch 63/100, Loss: 0.2397	Epoch 97/100, Loss: 0.1398
	Epoch 30/100, Loss: 0.3030	Epoch 64/100, Loss: 0.2687	Epoch 98/100, Loss: 0.1302
	Epoch 31/100, Loss: 0.2994	Epoch 65/100, Loss: 0.2574	Epoch 99/100, Loss: 0.1275
	Epoch 32/100, Loss: 0.3136	Epoch 66/100, Loss: 0.2497	Epoch 100/100, Loss: 0.1267
	Epoch 33/100, Loss: 0.3181	Epoch 67/100, Loss: 0.2415	Training complete.
	Epoch 34/100, Loss: 0.3079	Epoch 68/100, Loss: 0.2502	

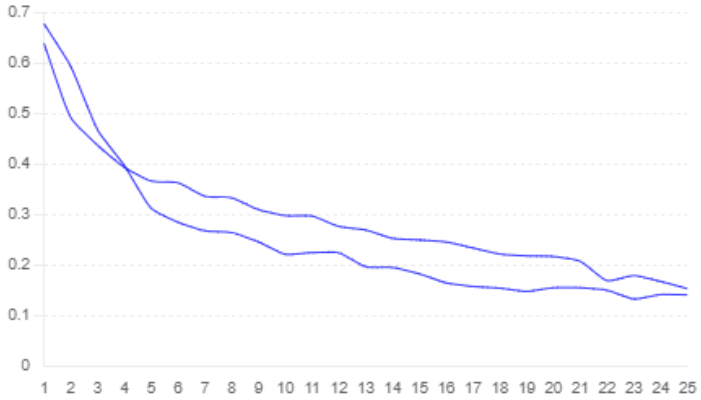


VGG16 (Pre-trained O)

Epoch=25	Epoch=100		
Epoch 1/25, Loss: 0.6386	Epoch 1/100, Loss: 0.6776	Epoch 35/100, Loss: 0.1195	Epoch 69/100, Loss: 0.0822
Epoch 2/25, Loss: 0.4926	Epoch 2/100, Loss: 0.5096	Epoch 36/100, Loss: 0.1156	Epoch 70/100, Loss: 0.0874
Epoch 3/25, Loss: 0.3929	Epoch 3/100, Loss: 0.4081	Epoch 37/100, Loss: 0.1184	Epoch 71/100, Loss: 0.0758
Epoch 4/25, Loss: 0.3363	Epoch 4/100, Loss: 0.3467	Epoch 38/100, Loss: 0.1183	Epoch 72/100, Loss: 0.0745
Epoch 5/25, Loss: 0.3133	Epoch 5/100, Loss: 0.3132	Epoch 39/100, Loss: 0.1256	Epoch 73/100, Loss: 0.0681
Epoch 6/25, Loss: 0.2834	Epoch 6/100, Loss: 0.2825	Epoch 40/100, Loss: 0.1214	Epoch 74/100, Loss: 0.0641
Epoch 7/25, Loss: 0.2655	Epoch 7/100, Loss: 0.2617	Epoch 41/100, Loss: 0.1192	Epoch 75/100, Loss: 0.0641
Epoch 8/25, Loss: 0.2490	Epoch 8/100, Loss: 0.2435	Epoch 42/100, Loss: 0.1159	Epoch 76/100, Loss: 0.0636
Epoch 9/25, Loss: 0.2415	Epoch 9/100, Loss: 0.2307	Epoch 43/100, Loss: 0.1110	Epoch 77/100, Loss: 0.0661
Epoch 10/25, Loss: 0.2306	Epoch 10/100, Loss: 0.2248	Epoch 44/100, Loss: 0.1116	Epoch 78/100, Loss: 0.0667
Epoch 11/25, Loss: 0.2290	Epoch 11/100, Loss: 0.2124	Epoch 45/100, Loss: 0.1041	Epoch 79/100, Loss: 0.0644
Epoch 12/25, Loss: 0.2193	Epoch 12/100, Loss: 0.2031	Epoch 46/100, Loss: 0.1043	Epoch 80/100, Loss: 0.0661
Epoch 13/25, Loss: 0.2091	Epoch 13/100, Loss: 0.1910	Epoch 47/100, Loss: 0.0989	Epoch 81/100, Loss: 0.0617
Epoch 14/25, Loss: 0.2014	Epoch 14/100, Loss: 0.1964	Epoch 48/100, Loss: 0.1007	Epoch 82/100, Loss: 0.0581
Epoch 15/25, Loss: 0.1962	Epoch 15/100, Loss: 0.1948	Epoch 49/100, Loss: 0.0936	Epoch 83/100, Loss: 0.0550
Epoch 16/25, Loss: 0.1937	Epoch 16/100, Loss: 0.2046	Epoch 50/100, Loss: 0.0893	Epoch 84/100, Loss: 0.0593
Epoch 17/25, Loss: 0.1905	Epoch 17/100, Loss: 0.2007	Epoch 51/100, Loss: 0.0975	Epoch 85/100, Loss: 0.0588
Epoch 18/25, Loss: 0.1872	Epoch 18/100, Loss: 0.1963	Epoch 52/100, Loss: 0.0953	Epoch 86/100, Loss: 0.0562
Epoch 19/25, Loss: 0.1835	Epoch 19/100, Loss: 0.1895	Epoch 53/100, Loss: 0.0915	Epoch 87/100, Loss: 0.0545
Epoch 20/25, Loss: 0.1763	Epoch 20/100, Loss: 0.1758	Epoch 54/100, Loss: 0.0872	Epoch 88/100, Loss: 0.0530
Epoch 21/25, Loss: 0.1738	Epoch 21/100, Loss: 0.1664	Epoch 55/100, Loss: 0.0874	Epoch 89/100, Loss: 0.0546
Epoch 22/25, Loss: 0.1693	Epoch 22/100, Loss: 0.1624	Epoch 56/100, Loss: 0.0856	Epoch 90/100, Loss: 0.0535
Epoch 23/25, Loss: 0.1648	Epoch 23/100, Loss: 0.1547	Epoch 57/100, Loss: 0.0824	Epoch 91/100, Loss: 0.0533
Epoch 24/25, Loss: 0.1605	Epoch 24/100, Loss: 0.1541	Epoch 58/100, Loss: 0.0826	Epoch 92/100, Loss: 0.0558
Epoch 25/25, Loss: 0.1531	Epoch 25/100, Loss: 0.1498	Epoch 59/100, Loss: 0.0801	Epoch 93/100, Loss: 0.0490
Training complete.	Epoch 26/100, Loss: 0.1460	Epoch 60/100, Loss: 0.0777	Epoch 94/100, Loss: 0.0537
	Epoch 27/100, Loss: 0.1441	Epoch 61/100, Loss: 0.0726	Epoch 95/100, Loss: 0.0626
	Epoch 28/100, Loss: 0.1422	Epoch 62/100, Loss: 0.0703	Epoch 96/100, Loss: 0.1135
	Epoch 29/100, Loss: 0.1316	Epoch 63/100, Loss: 0.0716	Epoch 97/100, Loss: 0.1067
	Epoch 30/100, Loss: 0.1320	Epoch 64/100, Loss: 0.0752	Epoch 98/100, Loss: 0.0863
	Epoch 31/100, Loss: 0.1328	Epoch 65/100, Loss: 0.0716	Epoch 99/100, Loss: 0.0761
	Epoch 32/100, Loss: 0.1271	Epoch 66/100, Loss: 0.0772	Epoch 100/100, Loss: 0.0685
	Epoch 33/100, Loss: 0.1238	Epoch 67/100, Loss: 0.0855	Training complete.
	Epoch 34/100, Loss: 0.1196	Epoch 68/100, Loss: 0.0836	

VGG16 (Pre-trained X)

Epoch=25	Epoch=100		
Epoch 1/25, Loss: 0.7234	Epoch 1/100, Loss: 0.7605	Epoch 35/100, Loss: 0.2240	Epoch 69/100, Loss: 0.1542
Epoch 2/25, Loss: 0.6239	Epoch 2/100, Loss: 0.6641	Epoch 36/100, Loss: 0.2092	Epoch 70/100, Loss: 0.1869
Epoch 3/25, Loss: 0.5497	Epoch 3/100, Loss: 0.5858	Epoch 37/100, Loss: 0.1987	Epoch 71/100, Loss: 0.1774
Epoch 4/25, Loss: 0.4849	Epoch 4/100, Loss: 0.5328	Epoch 38/100, Loss: 0.1893	Epoch 72/100, Loss: 0.1436
Epoch 5/25, Loss: 0.4491	Epoch 5/100, Loss: 0.4786	Epoch 39/100, Loss: 0.1950	Epoch 73/100, Loss: 0.1224
Epoch 6/25, Loss: 0.4233	Epoch 6/100, Loss: 0.4627	Epoch 40/100, Loss: 0.1799	Epoch 74/100, Loss: 0.1121
Epoch 7/25, Loss: 0.4104	Epoch 7/100, Loss: 0.4348	Epoch 41/100, Loss: 0.1735	Epoch 75/100, Loss: 0.1085
Epoch 8/25, Loss: 0.3900	Epoch 8/100, Loss: 0.4087	Epoch 42/100, Loss: 0.1692	Epoch 76/100, Loss: 0.1283
Epoch 9/25, Loss: 0.3751	Epoch 9/100, Loss: 0.3899	Epoch 43/100, Loss: 0.1915	Epoch 77/100, Loss: 0.1157
Epoch 10/25, Loss: 0.3585	Epoch 10/100, Loss: 0.3769	Epoch 44/100, Loss: 0.1789	Epoch 78/100, Loss: 0.1072
Epoch 11/25, Loss: 0.3858	Epoch 11/100, Loss: 0.3817	Epoch 45/100, Loss: 0.1893	Epoch 79/100, Loss: 0.1154
Epoch 12/25, Loss: 0.3589	Epoch 12/100, Loss: 0.3669	Epoch 46/100, Loss: 0.1956	Epoch 80/100, Loss: 0.1151
Epoch 13/25, Loss: 0.3542	Epoch 13/100, Loss: 0.3587	Epoch 47/100, Loss: 0.1725	Epoch 81/100, Loss: 0.1052
Epoch 14/25, Loss: 0.3385	Epoch 14/100, Loss: 0.3340	Epoch 48/100, Loss: 0.1604	Epoch 82/100, Loss: 0.1095
Epoch 15/25, Loss: 0.3320	Epoch 15/100, Loss: 0.3409	Epoch 49/100, Loss: 0.1533	Epoch 83/100, Loss: 0.1108
Epoch 16/25, Loss: 0.3367	Epoch 16/100, Loss: 0.3507	Epoch 50/100, Loss: 0.1481	Epoch 84/100, Loss: 0.1087
Epoch 17/25, Loss: 0.3407	Epoch 17/100, Loss: 0.3304	Epoch 51/100, Loss: 0.1431	Epoch 85/100, Loss: 0.1032
Epoch 18/25, Loss: 0.3090	Epoch 18/100, Loss: 0.3228	Epoch 52/100, Loss: 0.1368	Epoch 86/100, Loss: 0.1064
Epoch 19/25, Loss: 0.3038	Epoch 19/100, Loss: 0.3105	Epoch 53/100, Loss: 0.1315	Epoch 87/100, Loss: 0.0970
Epoch 20/25, Loss: 0.3117	Epoch 20/100, Loss: 0.2988	Epoch 54/100, Loss: 0.1339	Epoch 88/100, Loss: 0.0939
Epoch 21/25, Loss: 0.2968	Epoch 21/100, Loss: 0.3173	Epoch 55/100, Loss: 0.1294	Epoch 89/100, Loss: 0.0892
Epoch 22/25, Loss: 0.2791	Epoch 22/100, Loss: 0.3013	Epoch 56/100, Loss: 0.1331	Epoch 90/100, Loss: 0.0863
Epoch 23/25, Loss: 0.2766	Epoch 23/100, Loss: 0.2877	Epoch 57/100, Loss: 0.1898	Epoch 91/100, Loss: 0.0829
Epoch 24/25, Loss: 0.2739	Epoch 24/100, Loss: 0.3735	Epoch 58/100, Loss: 0.1737	Epoch 92/100, Loss: 0.0804
Epoch 25/25, Loss: 0.2568	Epoch 25/100, Loss: 0.3449	Epoch 59/100, Loss: 0.1443	Epoch 93/100, Loss: 0.0767
Training complete.	Epoch 26/100, Loss: 0.3057	Epoch 60/100, Loss: 0.1387	Epoch 94/100, Loss: 0.0834
	Epoch 27/100, Loss: 0.2940	Epoch 61/100, Loss: 0.1312	Epoch 95/100, Loss: 0.0780
	Epoch 28/100, Loss: 0.3074	Epoch 62/100, Loss: 0.1286	Epoch 96/100, Loss: 0.0756
	Epoch 29/100, Loss: 0.2798	Epoch 63/100, Loss: 0.1216	Epoch 97/100, Loss: 0.0768
	Epoch 30/100, Loss: 0.2607	Epoch 64/100, Loss: 0.1212	Epoch 98/100, Loss: 0.0865
	Epoch 31/100, Loss: 0.2514	Epoch 65/100, Loss: 0.1084	Epoch 99/100, Loss: 0.0890
	Epoch 32/100, Loss: 0.2401	Epoch 66/100, Loss: 0.1128	Epoch 100/100, Loss: 0.0870
	Epoch 33/100, Loss: 0.2405	Epoch 67/100, Loss: 0.1139	Training complete.
	Epoch 34/100, Loss: 0.2416	Epoch 68/100, Loss: 0.1818	



UNet

Epoch=25	Epoch=100		
Epoch 1/25, Loss: 0.6906	Epoch 1/100, Loss: 0.6748	Epoch 35/100, Loss: 0.3421	Epoch 69/100, Loss: 0.3181
Epoch 2/25, Loss: 0.6832	Epoch 2/100, Loss: 0.6699	Epoch 36/100, Loss: 0.3344	Epoch 70/100, Loss: 0.3169
Epoch 3/25, Loss: 0.6707	Epoch 3/100, Loss: 0.6640	Epoch 37/100, Loss: 0.3446	Epoch 71/100, Loss: 0.3839
Epoch 4/25, Loss: 0.6456	Epoch 4/100, Loss: 0.6483	Epoch 38/100, Loss: 0.3322	Epoch 72/100, Loss: 0.3346
Epoch 5/25, Loss: 0.5936	Epoch 5/100, Loss: 0.5734	Epoch 39/100, Loss: 0.3226	Epoch 73/100, Loss: 0.3203
Epoch 6/25, Loss: 0.5221	Epoch 6/100, Loss: 0.5198	Epoch 40/100, Loss: 0.3786	Epoch 74/100, Loss: 0.3197
Epoch 7/25, Loss: 0.5009	Epoch 7/100, Loss: 0.5207	Epoch 41/100, Loss: 0.3622	Epoch 75/100, Loss: 0.3060
Epoch 8/25, Loss: 0.4854	Epoch 8/100, Loss: 0.4890	Epoch 42/100, Loss: 0.3625	Epoch 76/100, Loss: 0.2926
Epoch 9/25, Loss: 0.4782	Epoch 9/100, Loss: 0.5372	Epoch 43/100, Loss: 0.3493	Epoch 77/100, Loss: 0.3130
Epoch 10/25, Loss: 0.4360	Epoch 10/100, Loss: 0.4782	Epoch 44/100, Loss: 0.3614	Epoch 78/100, Loss: 0.3448
Epoch 11/25, Loss: 0.4397	Epoch 11/100, Loss: 0.4567	Epoch 45/100, Loss: 0.3425	Epoch 79/100, Loss: 0.3224
Epoch 12/25, Loss: 0.4576	Epoch 12/100, Loss: 0.4360	Epoch 46/100, Loss: 0.3284	Epoch 80/100, Loss: 0.3043
Epoch 13/25, Loss: 0.4188	Epoch 13/100, Loss: 0.4213	Epoch 47/100, Loss: 0.4126	Epoch 81/100, Loss: 0.2961
Epoch 14/25, Loss: 0.4003	Epoch 14/100, Loss: 0.4199	Epoch 48/100, Loss: 0.3583	Epoch 82/100, Loss: 0.3043
Epoch 15/25, Loss: 0.4116	Epoch 15/100, Loss: 0.3987	Epoch 49/100, Loss: 0.3449	Epoch 83/100, Loss: 0.3093
Epoch 16/25, Loss: 0.3862	Epoch 16/100, Loss: 0.3875	Epoch 50/100, Loss: 0.3495	Epoch 84/100, Loss: 0.3268
Epoch 17/25, Loss: 0.3811	Epoch 17/100, Loss: 0.3829	Epoch 51/100, Loss: 0.3278	Epoch 85/100, Loss: 0.3110
Epoch 18/25, Loss: 0.3875	Epoch 18/100, Loss: 0.4363	Epoch 52/100, Loss: 0.3153	Epoch 86/100, Loss: 0.3088
Epoch 19/25, Loss: 0.3825	Epoch 19/100, Loss: 0.4071	Epoch 53/100, Loss: 0.3096	Epoch 87/100, Loss: 0.2917
Epoch 20/25, Loss: 0.3760	Epoch 20/100, Loss: 0.3828	Epoch 54/100, Loss: 0.3046	Epoch 88/100, Loss: 0.2826
Epoch 21/25, Loss: 0.3608	Epoch 21/100, Loss: 0.3727	Epoch 55/100, Loss: 0.2983	Epoch 89/100, Loss: 0.2726
Epoch 22/25, Loss: 0.3633	Epoch 22/100, Loss: 0.3745	Epoch 56/100, Loss: 0.4165	Epoch 90/100, Loss: 0.2688
Epoch 23/25, Loss: 0.3611	Epoch 23/100, Loss: 0.3932	Epoch 57/100, Loss: 0.3661	Epoch 91/100, Loss: 0.2904
Epoch 24/25, Loss: 0.3748	Epoch 24/100, Loss: 0.3624	Epoch 58/100, Loss: 0.3492	Epoch 92/100, Loss: 0.2778
Epoch 25/25, Loss: 0.3575	Epoch 25/100, Loss: 0.3677	Epoch 59/100, Loss: 0.3344	Epoch 93/100, Loss: 0.2759
Training complete.	Epoch 26/100, Loss: 0.4203	Epoch 60/100, Loss: 0.3401	Epoch 94/100, Loss: 0.2756
	Epoch 27/100, Loss: 0.3730	Epoch 61/100, Loss: 0.3291	Epoch 95/100, Loss: 0.2727
	Epoch 28/100, Loss: 0.3480	Epoch 62/100, Loss: 0.3326	Epoch 96/100, Loss: 0.2814
	Epoch 29/100, Loss: 0.3630	Epoch 63/100, Loss: 0.3284	Epoch 97/100, Loss: 0.2793
	Epoch 30/100, Loss: 0.3988	Epoch 64/100, Loss: 0.3143	Epoch 98/100, Loss: 0.2731
	Epoch 31/100, Loss: 0.3669	Epoch 65/100, Loss: 0.3256	Epoch 99/100, Loss: 0.2742
	Epoch 32/100, Loss: 0.3794	Epoch 66/100, Loss: 0.3133	Epoch 100/100, Loss: 0.2704
	Epoch 33/100, Loss: 0.3629	Epoch 67/100, Loss: 0.3142	Training complete.
	Epoch 34/100, Loss: 0.3499	Epoch 68/100, Loss: 0.3128	

Implementation

FCN

Implementation

VGG16 (Pre-trained O)

Implementation

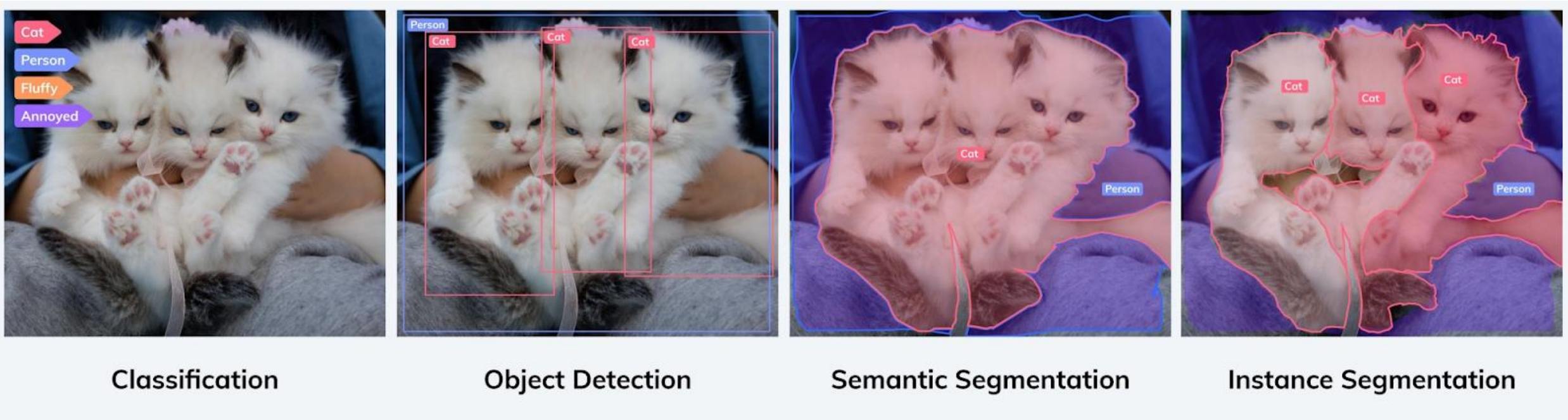
VGG16 (Pre-trained X)

Implementation

UNet

Mask R-CNN

Introduction



Classification | Object Detection | Semantic Segmentation | Instance Segmentation

Introduction

Instance Segmentation

Correct detection of all objects + Precisely segmenting each instance

Faster R-CNN + Mask branch

Adding a branch for predicting segmentation masks on each RoI
in parallel with existing branch for classification and bbox regression

RoIAlign

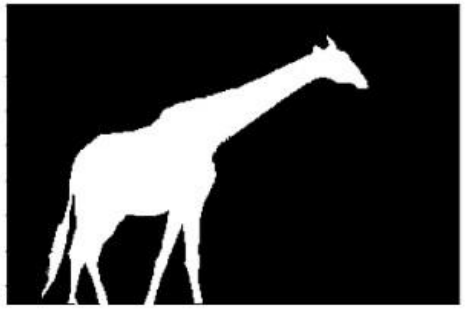
Simple, quantization-free layer that preserves exact spatial locations
(∵ Faster R-CNN [RPN] was not designed for pixel-to-pixel alignment)

→ Improves mask accuracy 10% to 50%

Introduction

Base

Binary Mask



"Without bells and whistles"

Mask R-CNN surpasses all previous SOTA single-model results on COCO instance segmentation task without adding additional supplementary functions.

	FCN	Mask R-CNN
Perform	Per-pixel multi-class categorization	Binary mask
Segmentation & Classification	Couple	Decouple (Predict binary mask for each class) → Rely on network's RoI classification

Related Work

R-CNN

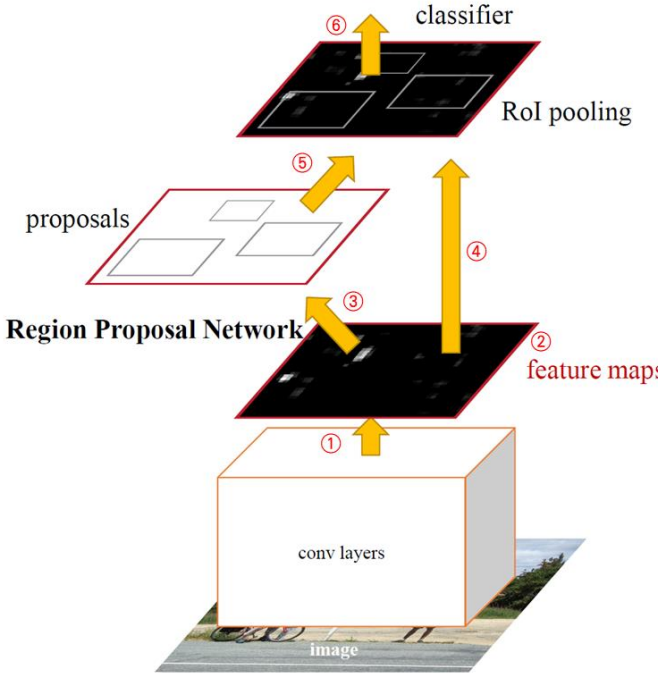
RoI-Pooling

$$\frac{\partial L}{\partial x_i} = \sum_r \sum_j [i = i^*(r,j)] \frac{\partial L}{\partial y_{rj}}$$

$$y_{rj} = x_{i^*(r,j)}$$

$$i^*(r,j) = argmax_{i' \in \mathcal{R}(r,j)} x_{i'}$$

RPN



Robust
Flexible

Instance Segmentation

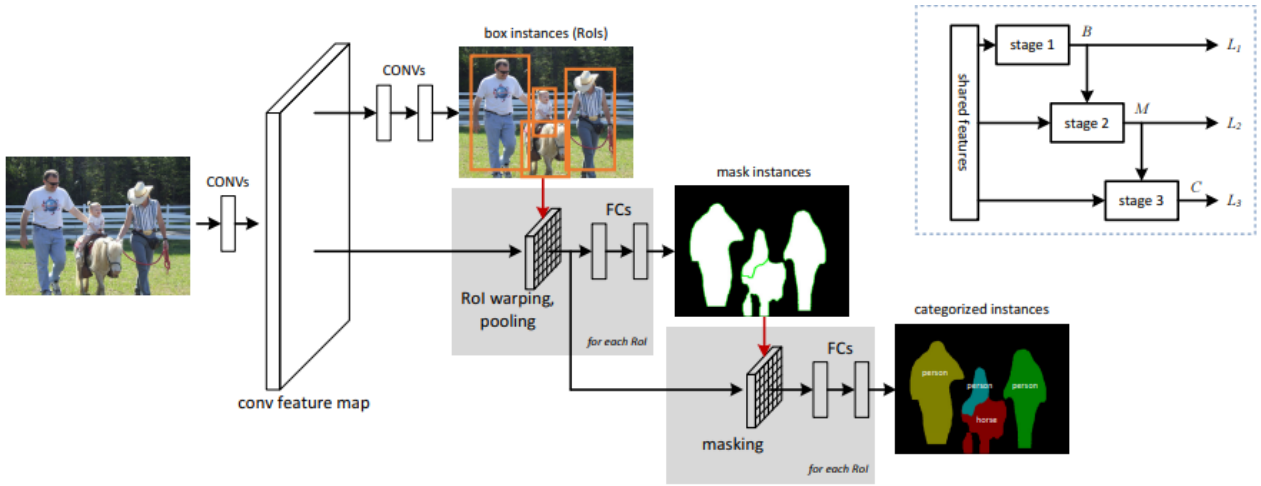


Figure 2. Multi-task Network Cascades for instance-aware semantic segmentation. At the top right corner is a simplified illustration.

Multiple-Stage ⇒ parallel

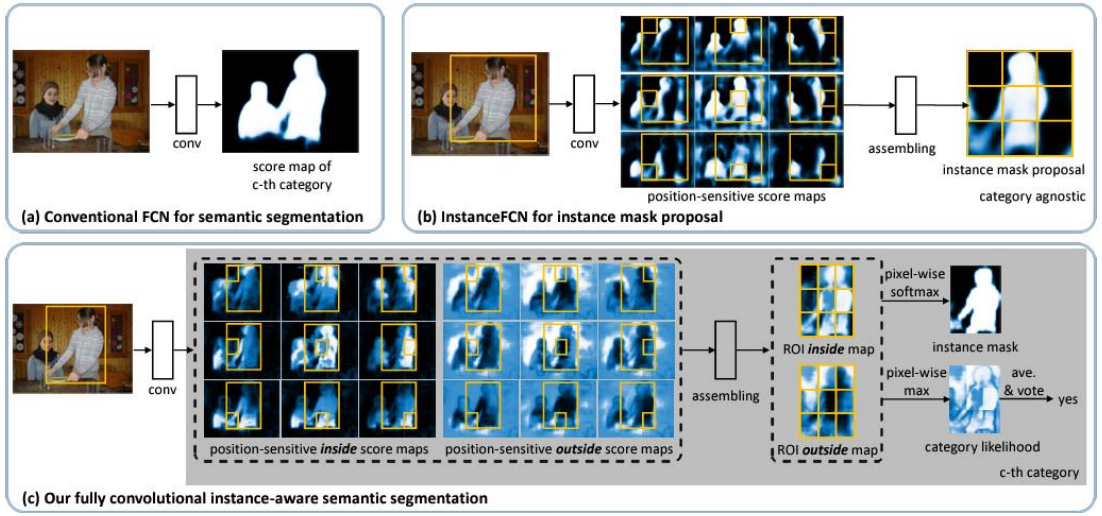
Simple
Flexible

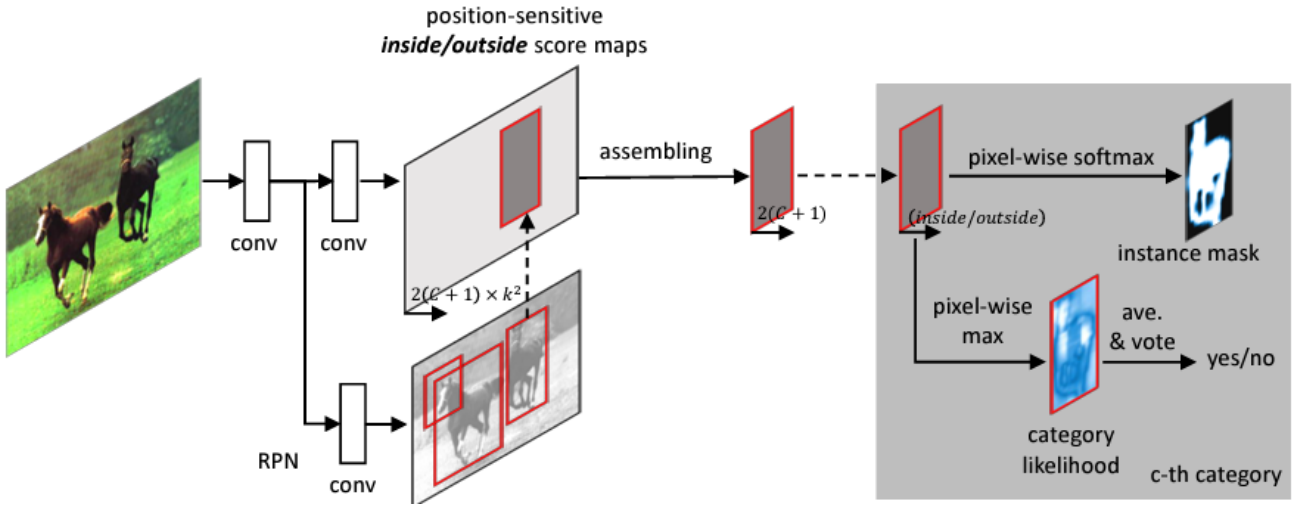
*Jifeng Dai, Kaiming He, Jain Sun.「**Instance-aware Semantic Segmentation via Multi-task Network Cascades**」, 2016*

Related Work

Instance Segmentation

Instance-aware Semantic Segmentation





Segment proposal system + Object detection system

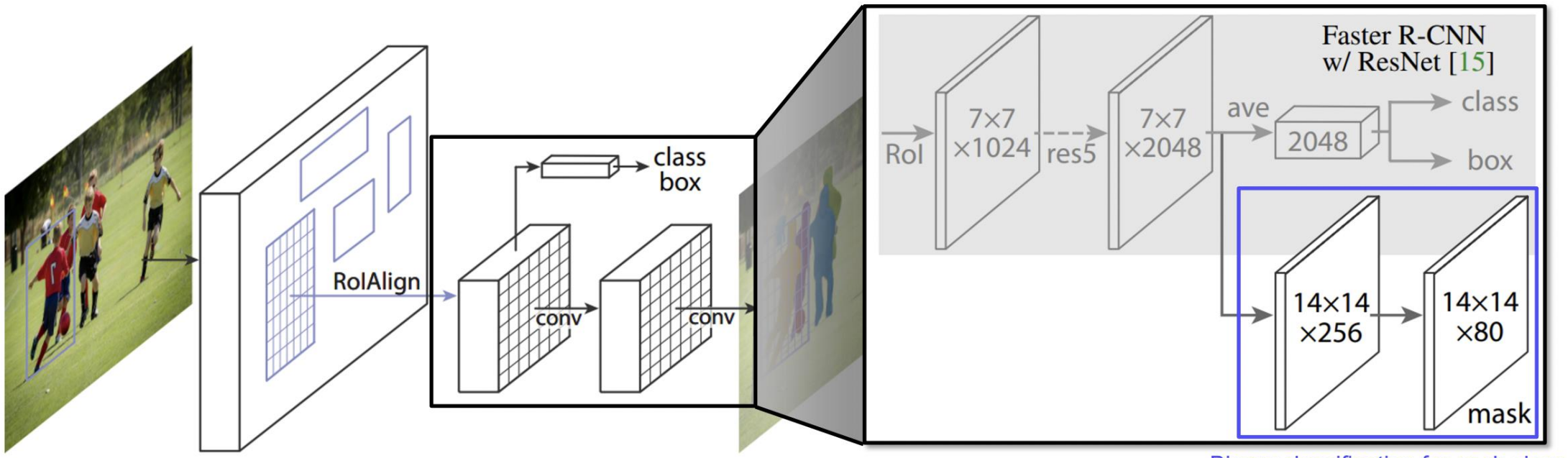
In order to predict a set of position sensitive output channels fully convolutionally

Systematic errors on overlapping instances and creates spurious edges

*Jifeng Dai, Yi Li, Haozhi Qi, Xiangyang Ji, Yichen Wei.「**Fully Convolutional Instance-aware Semantic Segmentation**」, 2017*

Mask R-CNN

Faster R-CNN + Mask branch



Mask R-CNN = Faster R-CNN + Mask branch

Mask R-CNN

RoIAlign – remove harsh quantization of RoIPool

0.8	1.1	2.1	3.9	0.4
2.2	4.3	4.2	1.8	6.3
0.2	5.5	4.8	7.7	9.5
1.1	0.8	2.2	0.8	5.2
5.0	1.8	1.9	2.0	4.1

Quantization

0.8	1.1	2.1	3.9	0.4
2.2	4.3	4.2	1.8	6.3
0.2	5.5	4.8	7.7	9.5
1.1	0.8	2.2	0.8	5.2
5.0	1.8	1.9	2.0	4.1

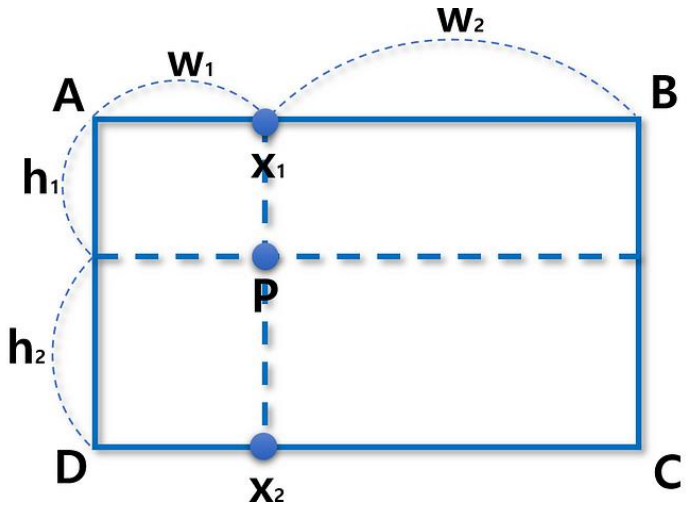
Cause **misalignment**
between RoI and extracted features

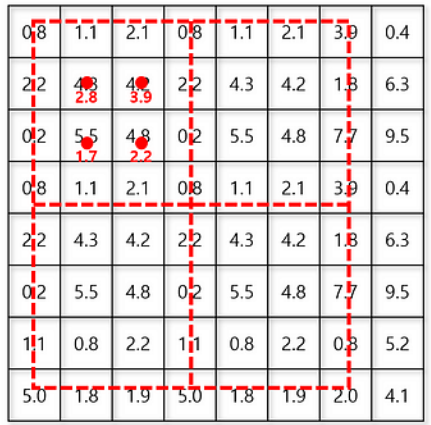
⇒ Negative effect on
predicting pixel-accurate mask

	AP^{kp}	AP^{kp}_{50}	AP^{kp}_{75}	AP^{kp}_{M}	AP^{kp}_{L}
RoIPool	59.8	86.2	66.7	55.1	67.4
RoIAlign	**64.2**	**86.6**	**69.7**	**58.7**	**73.0**

Table 6. **RoIAlign** *vs.* **RoIPool** for keypoint detection on `minival`.

"Bilinear Interpolation"





Mask R-CNN

Loss Function

$$L = L_{cls} + L_{box} + L_{mask}$$

L_{mask}

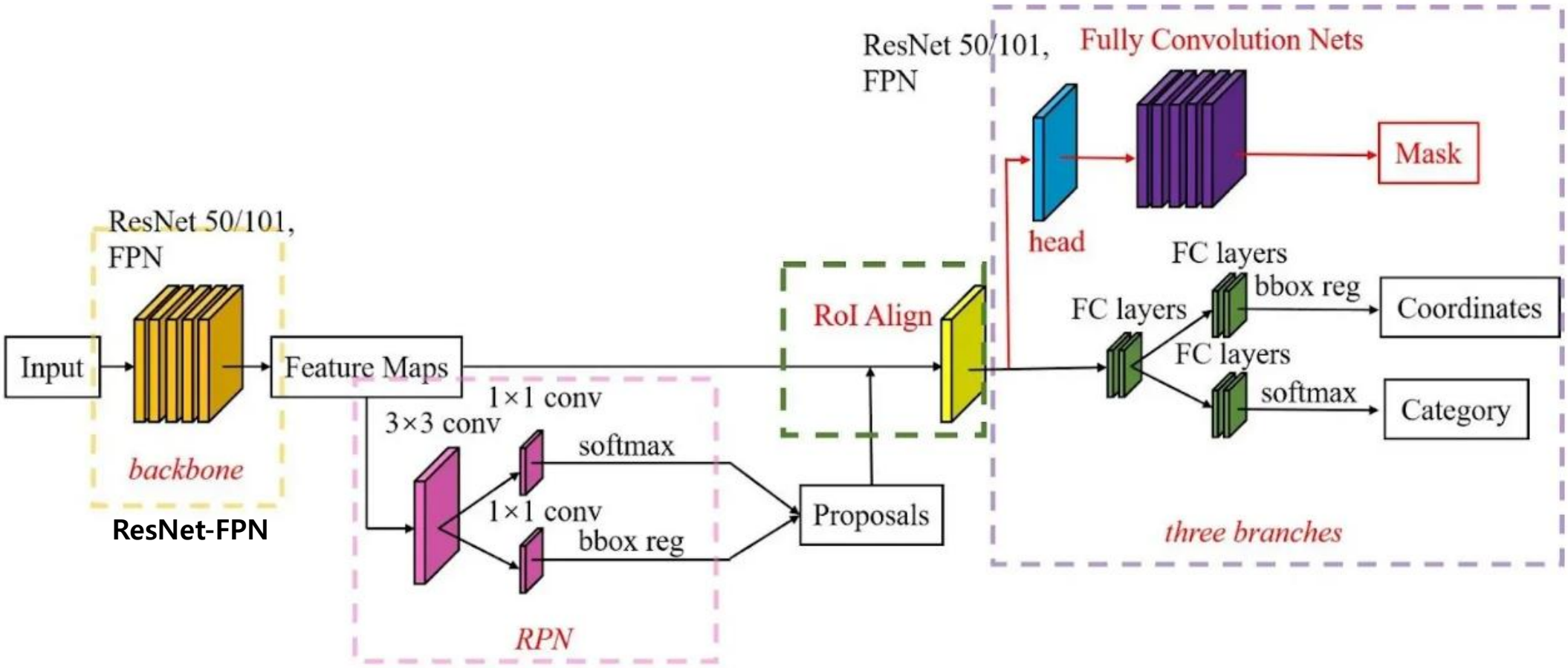
- km^2-dimension ($m \times m$ images $* k$ binary mask class)
- Apply per-pixel sigmoid
- Average binary cross-entropy loss

Predict $m * m$ mask from each RoI using FCN

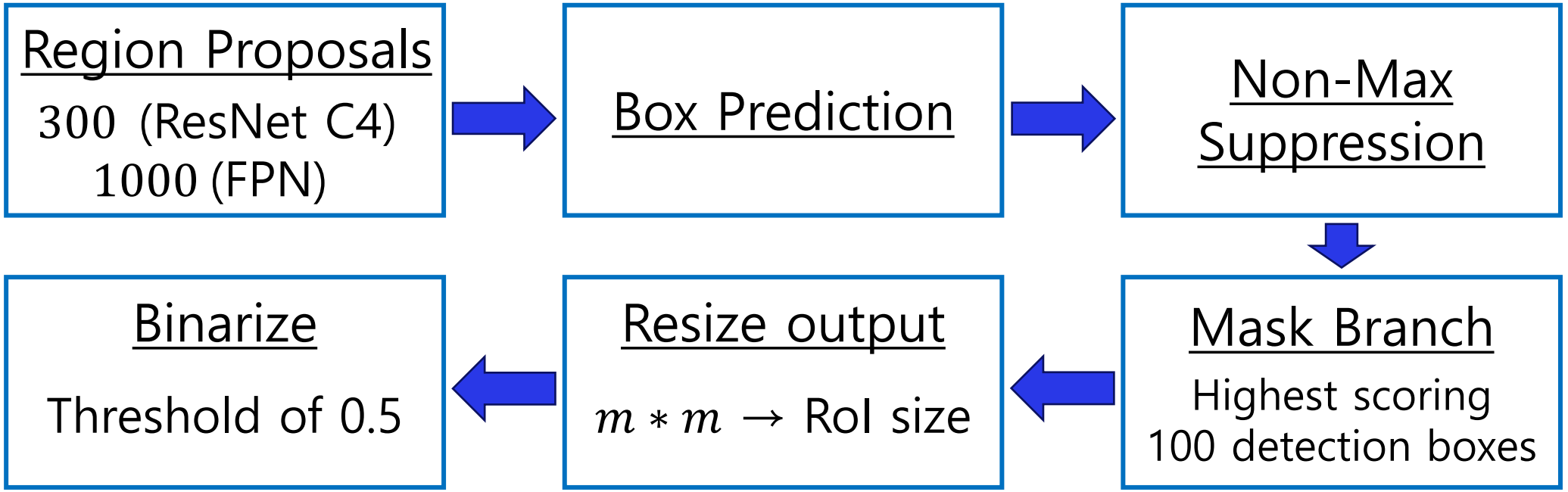
$\Rightarrow$ Allow each layer maintain explicit $m * m$ object spatial layout

Use convolution layer instead of fc layer

Architecture



Inference



Use k-th mask (predicted class)
instead of K per RoI (mask branch predict)

Ablations

net-depth-features	AP	AP_{50}	AP_{75}
ResNet-50-C4	30.3	51.2	31.5
ResNet-101-C4	32.7	54.2	34.3
ResNet-50-FPN	33.6	55.2	35.3
ResNet-101-FPN	35.4	57.3	37.5
ResNeXt-101-FPN	**36.7**	**59.5**	**38.9**

(a) **Backbone Architecture**: Better backbones bring expected gains: deeper networks do better, FPN outperforms C4 features, and ResNeXt improves on ResNet.

	AP	AP_{50}	AP_{75}
softmax	24.8	44.1	25.1
sigmoid	**30.3**	**51.2**	**31.5**
	+5.5	*+7.1*	*+6.4*

(b) **Multinomial *vs.* Independent Masks** (ResNet-50-C4): *Decoupling* via per-class binary masks (sigmoid) gives large gains over multinomial masks (softmax).

	align?	bilinear?	agg.	AP	AP_{50}	AP_{75}
RoIPool [9]			max	26.9	48.8	26.4
RoIWarp [7]		✓	max	27.2	49.2	27.1
		✓	ave	27.1	48.9	27.1
RoIAlign	✓	✓	max	**30.2**	**51.0**	**31.8**
	✓	✓	ave	**30.3**	**51.2**	**31.5**

(c) **RoIAlign** (ResNet-50-C4): Mask results with various RoI layers. Our RoIAlign layer improves AP by ~3 points and AP_{75} by ~5 points. Using proper alignment is the only factor that contributes to the large gap between RoI layers.

	AP	AP_{50}	AP_{75}	AP^{bb}	AP^{bb}_{50}	AP^{bb}_{75}
RoIPool	23.6	46.5	21.6	28.2	52.7	26.9
RoIAlign	**30.9**	**51.8**	**32.1**	**34.0**	**55.3**	**36.4**
	+7.3	*+ 5.3*	*+10.5*	*+5.8*	*+2.6*	*+9.5*

(d) **RoIAlign** (ResNet-50-**C5**, *stride 32*): Mask-level and box-level AP using *large-stride* features. Misalignments are more severe than with stride-16 features (Table 2c), resulting in massive accuracy gaps.

	mask branch	AP	AP_{50}	AP_{75}
MLP	fc: $1024 \rightarrow 1024 \rightarrow 80 \cdot 28^2$	31.5	53.7	32.8
MLP	fc: $1024 \rightarrow 1024 \rightarrow 1024 \rightarrow 80 \cdot 28^2$	31.5	54.0	32.6
FCN	conv: $256 \rightarrow 256 \rightarrow 256 \rightarrow 256 \rightarrow 256 \rightarrow 80$	**33.6**	**55.2**	**35.3**

(e) **Mask Branch** (ResNet-50-FPN): Fully convolutional networks (FCN) *vs.* multi-layer perceptrons (MLP, fully-connected) for mask prediction. FCNs improve results as they take advantage of explicitly encoding spatial layout.

Table 2. **Ablations** for Mask R-CNN. We train on `trainval35k`, test on `minival`, and report *mask* AP unless otherwise noted.

Mask R-CNN for Human Pose Estimation

Figure 6. Keypoint detection results on COCO `test` using Mask R-CNN (ResNet-50-FPN), with person segmentation masks predicted from the same model. This model has a keypoint AP of 63.1 and runs at 5 fps.

Keypoint detection

	AP^{kp}	AP^{kp}_{50}	AP^{kp}_{75}	AP^{kp}_{M}	AP^{kp}_{L}
CMU-Pose+++ [4]	61.8	84.9	67.5	57.1	68.2
G-RMI [26]†	62.4	84.0	68.5	**59.1**	68.1
Mask R-CNN, keypoint-only	62.7	87.0	68.4	57.4	71.1
Mask R-CNN, keypoint & mask	**63.1**	**87.3**	**68.7**	57.8	**71.4**

Multi-task learning

	AP^{bb}_{person}	AP^{mask}_{person}	AP^{kp}
Faster R-CNN	52.5	-	-
Mask R-CNN, mask-only	**53.6**	**45.8**	-
Mask R-CNN, keypoint-only	50.7	-	64.2
Mask R-CNN, keypoint & mask	52.0	45.1	**64.7**

Experiment

MS COCO

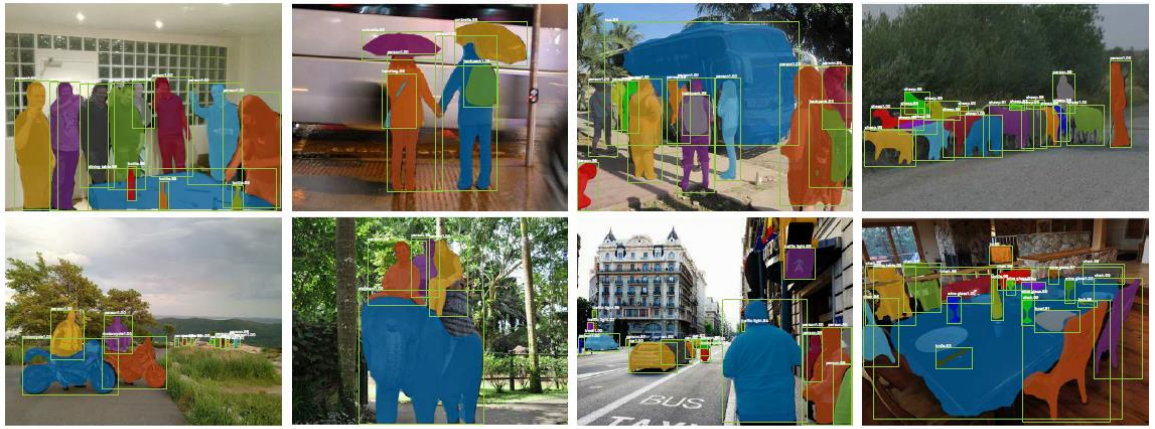


Figure 2. **Mask R-CNN** results on the COCO test set. These results are based on ResNet-101 [15], achieving a *mask* AP of 35.7 and running at 5 fps. Masks are shown in color, and bounding box, category, and confidences are also shown.

Figure 4. More results of **Mask R-CNN** on COCO test images, using ResNet-101-FPN and running at 5 fps, with 35.7 mask AP (Table 1).

	backbone	AP	AP_{50}	AP_{75}	AP_S	AP_M	AP_L
MNC [7]	ResNet-101-C4	24.6	44.3	24.8	4.7	25.9	43.6
FCIS [21] +OHEM	ResNet-101-C5-dilated	29.2	49.5	-	7.1	31.3	50.0
FCIS+++ [21] +OHEM	ResNet-101-C5-dilated	33.6	54.5	-	-	-	-
Mask R-CNN	ResNet-101-C4	33.1	54.9	34.8	12.1	35.6	51.1
Mask R-CNN	ResNet-101-FPN	35.7	58.0	37.8	15.5	38.1	52.4
Mask R-CNN	ResNeXt-101-FPN	**37.1**	**60.0**	**39.4**	**16.9**	**39.9**	**53.5**

Experiment

FCIS vs Mask R-CNN

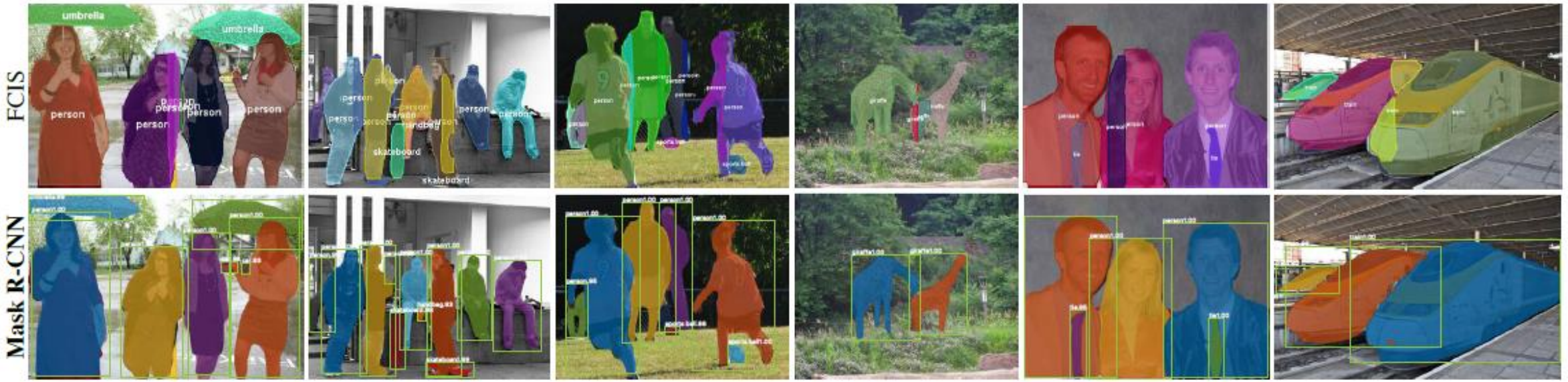
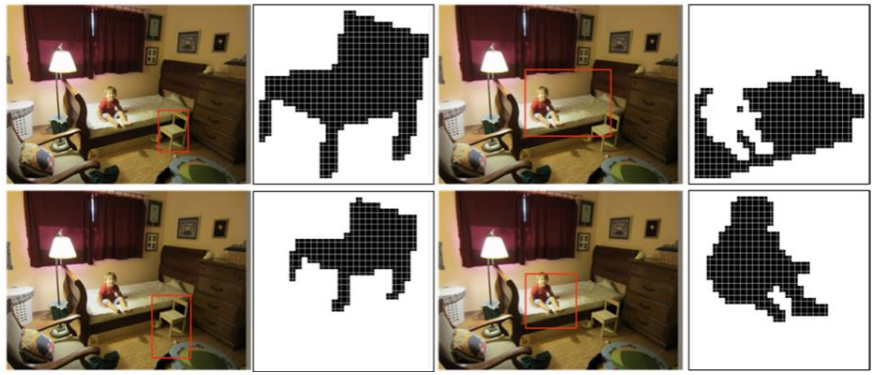


Figure 5. FCIS+++ [21] (top) *vs*. Mask R-CNN (bottom, ResNet-101-FPN). FCIS exhibits systematic artifacts on overlapping objects.

*Jifeng Dai, Yi Li, Haozhi Qi, Xiangyang Ji, Yichen Wei.「**Fully Convolutional Instance-aware Semantic Segmentation**」, 2017*

Result

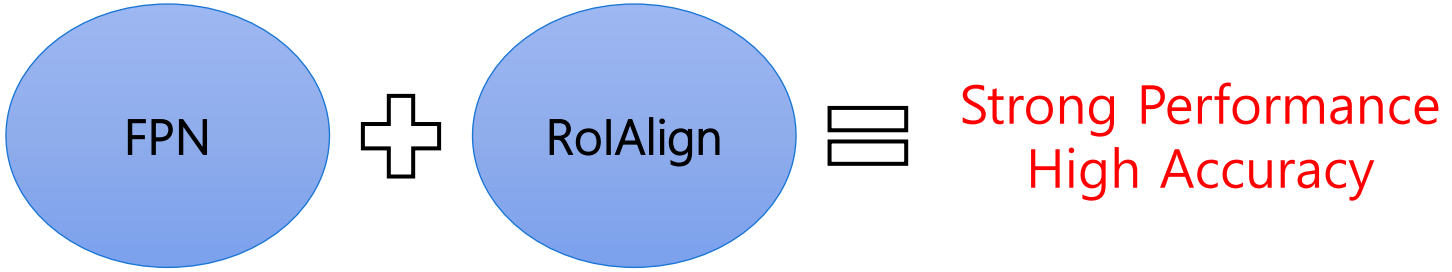


	training data	AP [val]	AP	AP_{50}	person	rider	car	truck	bus	train	mcycle	bicycle
InstanceCut [23]	fine + coarse	15.8	13.0	27.9	10.0	8.0	23.7	14.0	19.5	15.2	9.3	4.7
DWT [4]	fine	19.8	15.6	30.0	15.1	11.7	32.9	17.1	20.4	15.0	7.9	4.9
SAIS [17]	fine	-	17.4	36.7	14.6	12.9	35.7	16.0	23.2	19.0	10.3	7.8
DIN [3]	fine + coarse	-	20.0	38.8	16.5	16.7	25.7	20.6	30.0	23.4	17.1	10.1
SGN [29]	fine + coarse	29.2	25.0	44.9	21.8	20.1	39.4	24.8	33.2	30.8	17.7	12.4
Mask R-CNN	fine	31.5	26.2	49.9	30.5	23.7	46.9	22.8	32.2	18.6	19.1	16.0
Mask R-CNN	fine + COCO	**36.4**	**32.0**	**58.1**	**34.8**	**27.0**	**49.1**	**30.1**	**40.9**	**30.9**	**24.1**	**18.7**

Table 7: Results on Cityscapes `val` ('AP [`val`]' column) and `test` (remaining columns) sets. Our method uses ResNet-50-FPN.

Conclusion

Mask R-CNN = Object Detection + Instance Segmentation

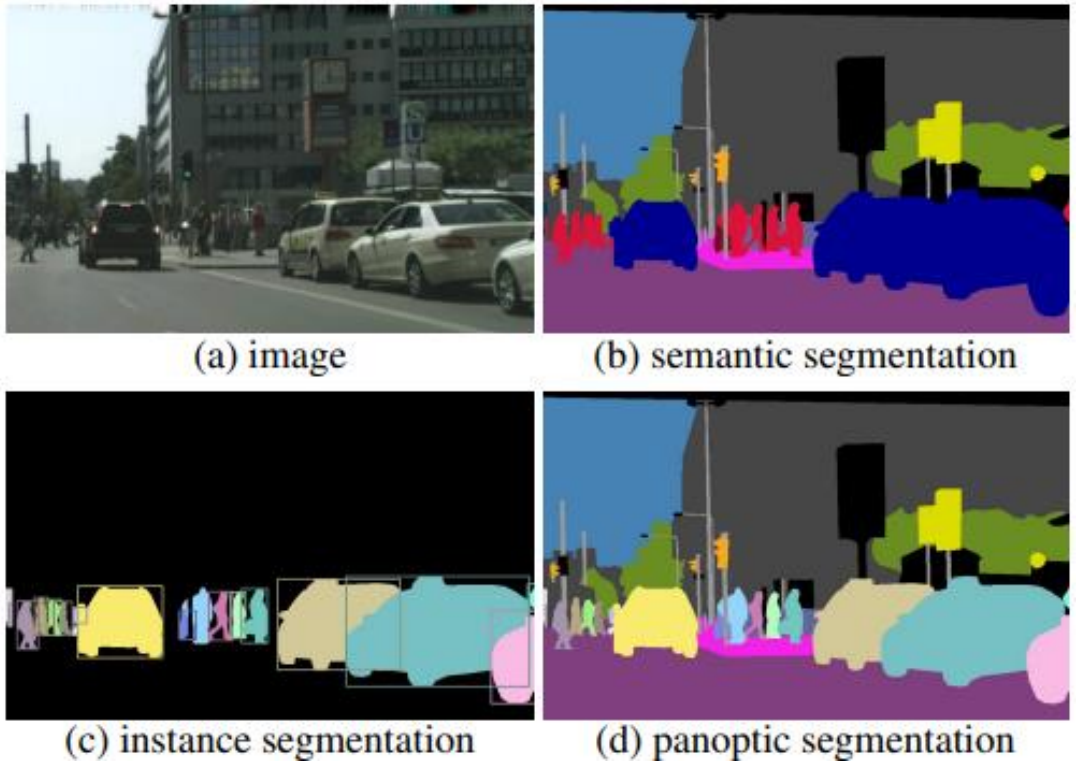


< Limitations >

- Computational Complexity
- Small-Object Segmentation
- Data Requirements
- Limited Generalization to Unseen Categories

Panoptic Segmentation

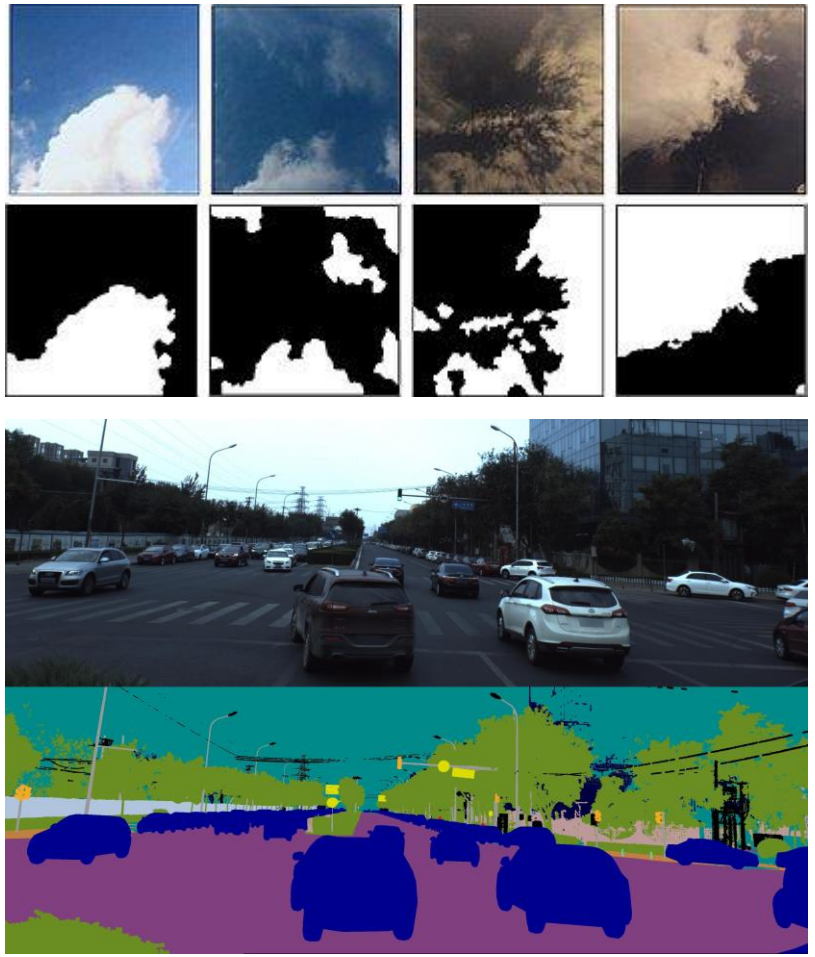
Introduction



Semantic Segmentation
(Assign class label to each pixel)
+
Instance Segmentation
(Detect and Segment each object instance)
=
Panoptic Segmentation
(Including everything visible in one view)

Lack of appropriate metrics or associated recognition challenges
⇒ **"Panoptic Quality (PQ)"**

Format



Task format

$$\mathcal{L} := \{0, \dots, L-1\}, \qquad (l_i, z_i) \in \mathcal{L} \times \mathbb{N}$$

- l_i : semantic class of pixel i
- z_i : instance id

Stuff and Thing labels

$$\mathcal{L} = \mathcal{L}^{St} \cup \mathcal{L}^{Th}, \qquad \mathcal{L}^{St} \cap \mathcal{L}^{Th} = \emptyset$$

- $l_i \in \mathcal{L}^{St}$: Stuff classes all pixels belong to same instance
  $\Leftrightarrow$ its corresponding instance id z_i is irrelevant

- $l_i \in \mathcal{L}^{Th}$: All pixels belonging to a single instance must have the same (l_i, z_i)

Metric

- **Completeness**
- **Interpretability**
- **Simplicity**

Segment Matching

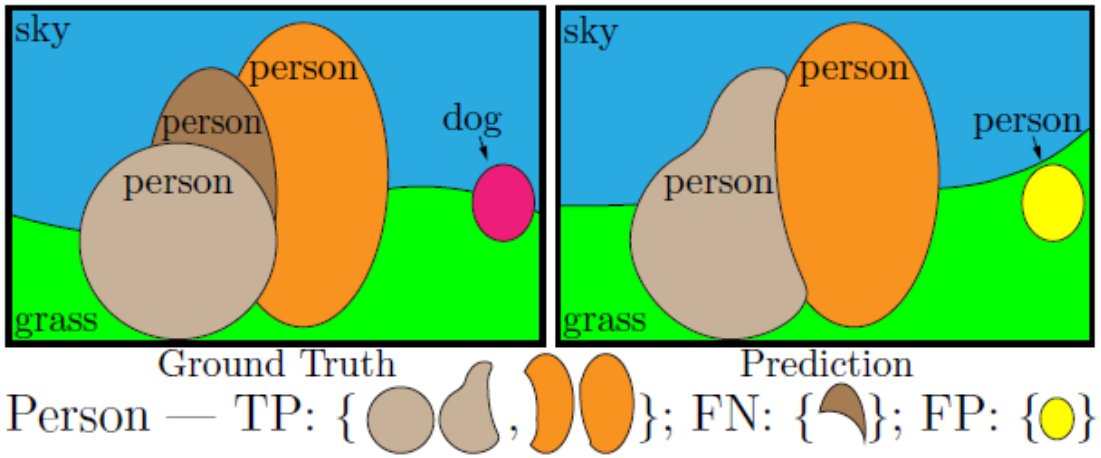
$$IoU(p_i, g) = \frac{|p_i \cap g|}{|p_i \cup g|} \leq \frac{|p_i \cap g|}{|g|} \quad \text{for} \quad i \in \{1,2\}$$

$$|p_1 \cap g| + |p_2 \cap g| \leq |g| \quad (\because p_1 \cap p_2 = \emptyset)$$

$$\Rightarrow IoU(p_1, g) + IoU(p_2, g) \leq \frac{|p_1 \cap g| + |p_2 \cap g|}{|g|} \leq 1$$

$$\therefore IoU(p_1, g) > 0.5 \leftrightarrow IoU(p_2, g) < 0.5$$

Only one ground truth segment can have IoU



PQ Computation

$$PQ = \frac{\sum_{(p,g) \in TP} IoU(p, g)}{|TP| + \frac{1}{2}|FP| + \frac{1}{2}|FN|}$$

Penalize segment without matches

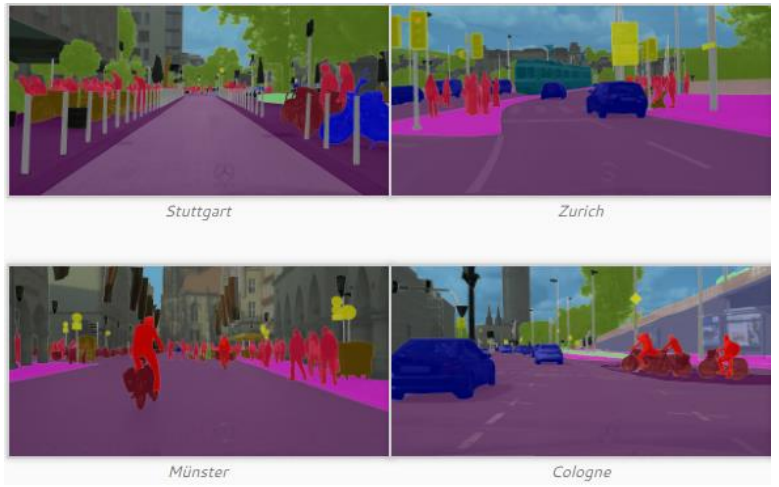
$\frac{\sum_{(p,g) \in TP} IoU(p,g)}{|TP|}$: avg IoU of matched segments

$$PQ = \underbrace{\frac{\sum_{(p,g) \in TP} IoU(p, g)}{|TP|}}_{\text{Segmentation Quality}} \times \underbrace{\frac{|TP|}{|TP| + \frac{1}{2}|FP| + \frac{1}{2}|FN|}}_{\text{Recognition Quality}}$$

Result

Datasets

- Cityscapes
  - 5000 images (2975 train, 500 val, 1525 test)
  - Dense pixel annotations (97% coverage) of 19 classes
- ADE20k
  - 25k↑ images (20k train, 2k val, 3k test)
  - Densely annotated with an open-dictionary label set
- Mapillary Vistas
  - 25k street-view images (18k train, 2k val, 5k test)
  - Densely annotated (98% pixel coverage)
    with 28 stuff and 37 thing classes



Human Consistency

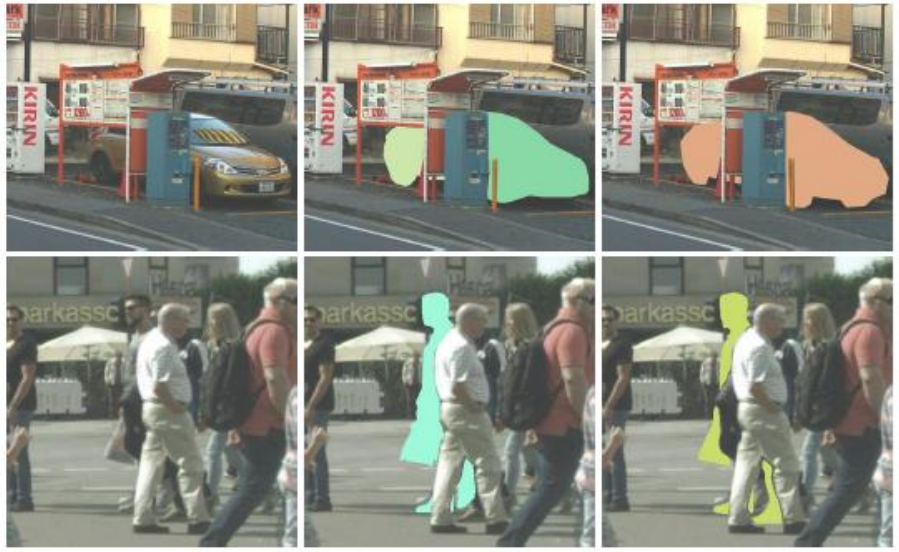


Figure 3: **Segmentation flaws.** Images are zoomed and cropped. Top row (Vistas image): both annotators identify the object as a car, however, one splits the car into two cars. Bottom row (Cityscapes image): the segmentation is genuinely ambiguous.

	PQ	PQ^{St}	PQ^{Th}	SQ	SQ^{St}	SQ^{Th}	RQ	RQ^{St}	RQ^{Th}
Cityscapes	69.7	71.3	67.4	84.2	84.4	83.9	82.1	83.4	80.2
ADE20k	67.1	70.3	65.9	85.8	85.5	85.9	78.0	82.4	76.4
Vistas	57.5	62.6	53.4	79.5	81.6	77.9	71.4	76.0	67.7

Table 1: **Human consistency for stuff *vs.* things.** Panoptic, segmentation, and recognition quality (PQ, SQ, RQ) averaged over classes (PQ=SQ×RQ per class) are reported as percentages. Perhaps surprisingly, we find that human consistency on each dataset is relatively similar for both stuff and things.

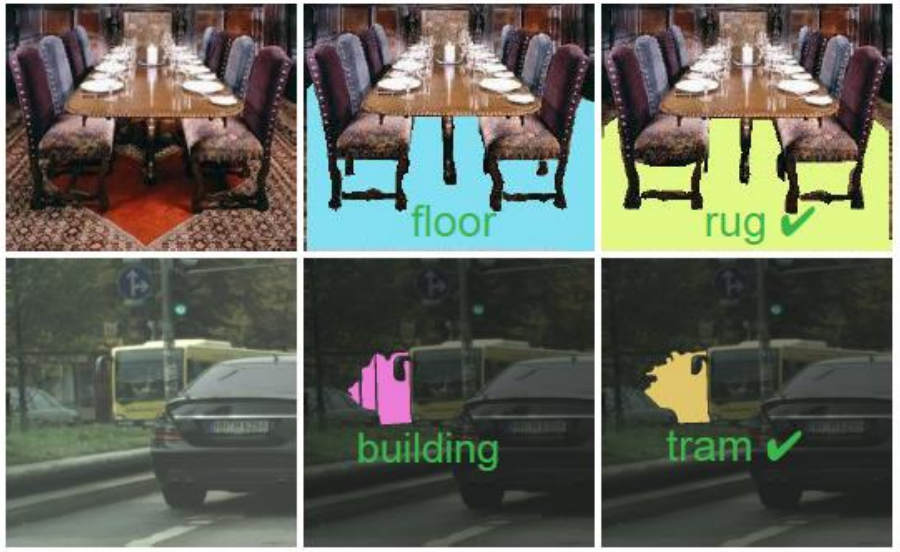


Figure 4: **Classification flaws.** Images are zoomed and cropped. Top row (ADE20k image): simple misclassification. Bottom row (Cityscapes image): the scene is extremely difficult, tram is the correct class for the segment. Many errors are difficult to resolve.

	PQ^{S}	PQ^{M}	PQ^{L}	SQ^{S}	SQ^{M}	SQ^{L}	RQ^{S}	RQ^{M}	RQ^{L}
Cityscapes	35.1	62.3	84.8	67.8	81.0	89.9	51.5	76.5	94.1
ADE20k	49.9	69.4	79.0	78.0	84.0	87.8	64.2	82.5	89.8
Vistas	35.6	47.7	69.4	70.1	76.6	83.1	51.5	62.3	82.6

Table 2: **Human consistency *vs.* scale**, for small (S), medium (M) and large (L) objects. Scale plays a large role in determining human consistency for panoptic segmentation. On large objects both SQ and RQ are above 80 on all datasets, while for small objects RQ drops precipitously. SQ for small objects is quite reasonable.

Human Consistency

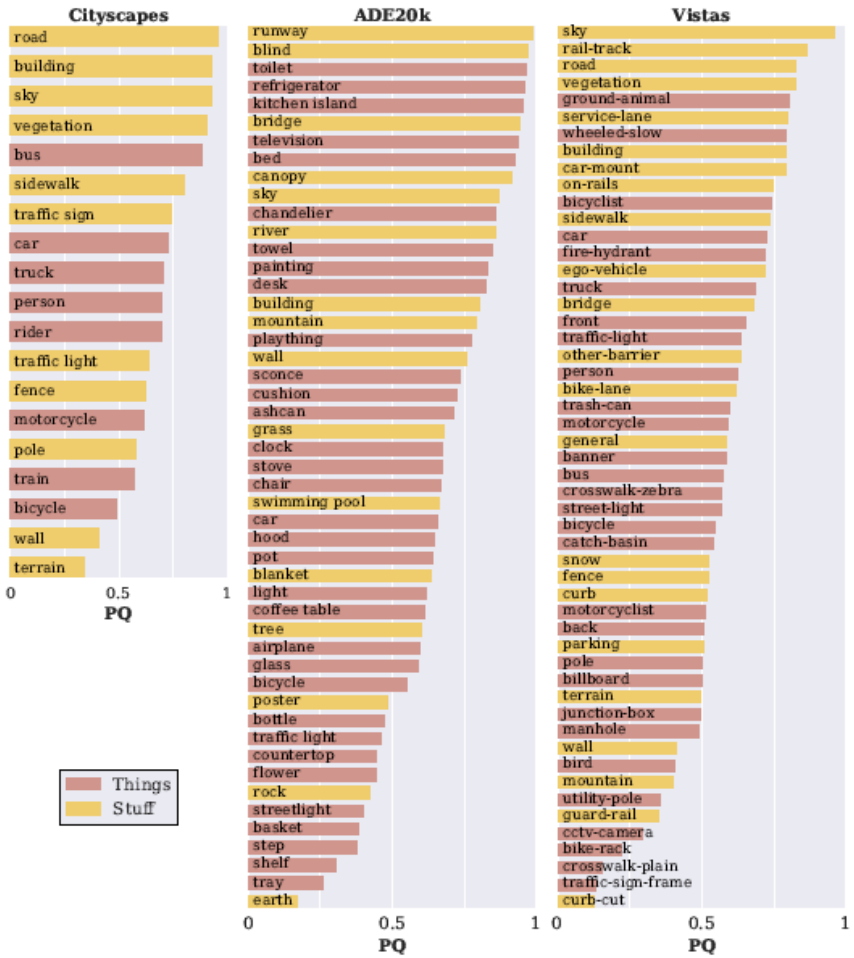


Figure 5: **Per-Class Human consistency, sorted by PQ**. Thing classes are shown in red, stuff classes in orange (for ADE20k every other class is shown, classes without matches in the dual-annotated tests sets are omitted). Things and stuff are distributed fairly evenly, implying PQ balances their performance.

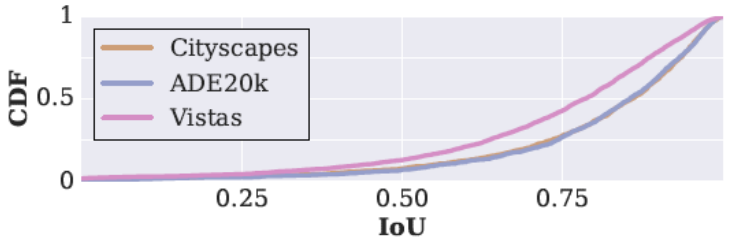


Figure 6: **Cumulative density functions of overlaps** for matched segments in three datasets when matches are computed by solving a maximum weighted bipartite matching problem [47]. After matching, less than 16% of matched objects have IoU below 0.5.

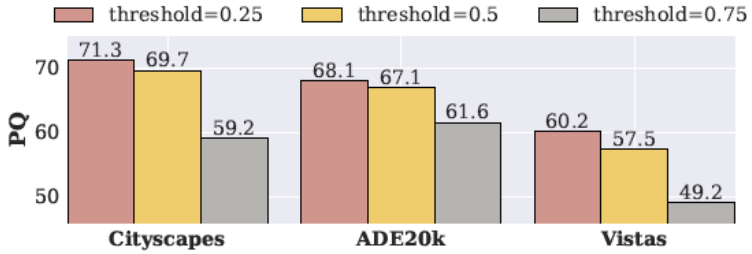


Figure 7: **Human consistency for different IoU thresholds.** The difference in PQ using a matching threshold of 0.25 *vs.* 0.5 is relatively small. For IoU of 0.25 matching is obtained by solving a maximum weighted bipartite matching problem. For a threshold greater than 0.5 the matching is unique and much easier to obtain.

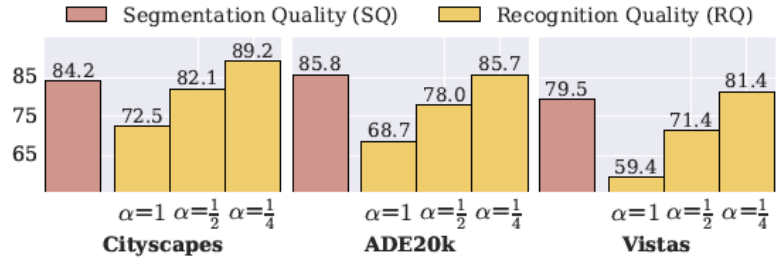


Figure 8: **SQ *vs.* RQ** for different α, see (3). Lowering α reduces the penalty of unmatched segments and thus increases the reported RQ (SQ is not affected). We use α of 0.5 throughout but by tuning α one can balance the influence of SQ and RQ in the final metric.

Result

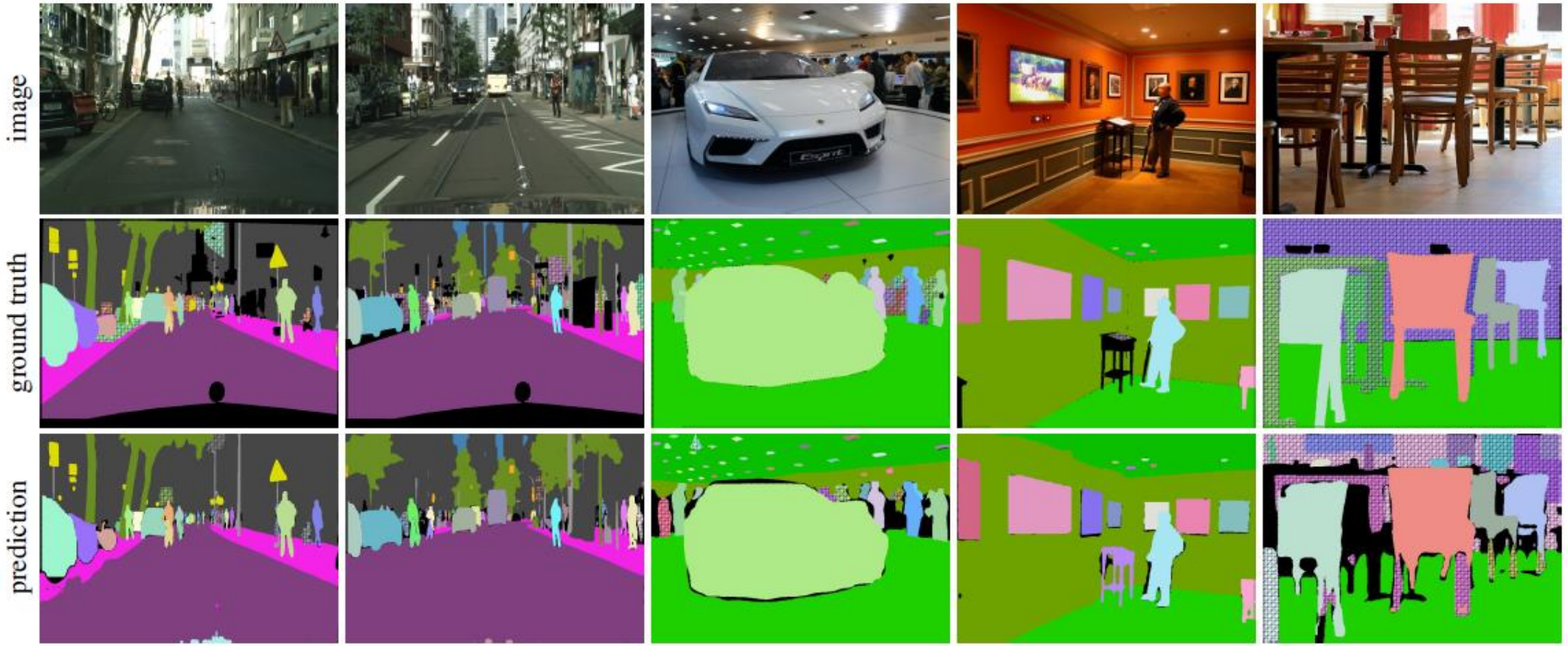


Figure 9: **Panoptic segmentation results** on Cityscapes (left two) and ADE20k (right three). Predictions are based on the merged outputs of state-of-the-art instance and semantic segmentation algorithms (see Tables 3 and 4). Colors for matched segments (IoU>0.5) match (crosshatch pattern indicates unmatched regions and black indicates unlabeled regions). Best viewed in color and with zoom.

Video Object Detection

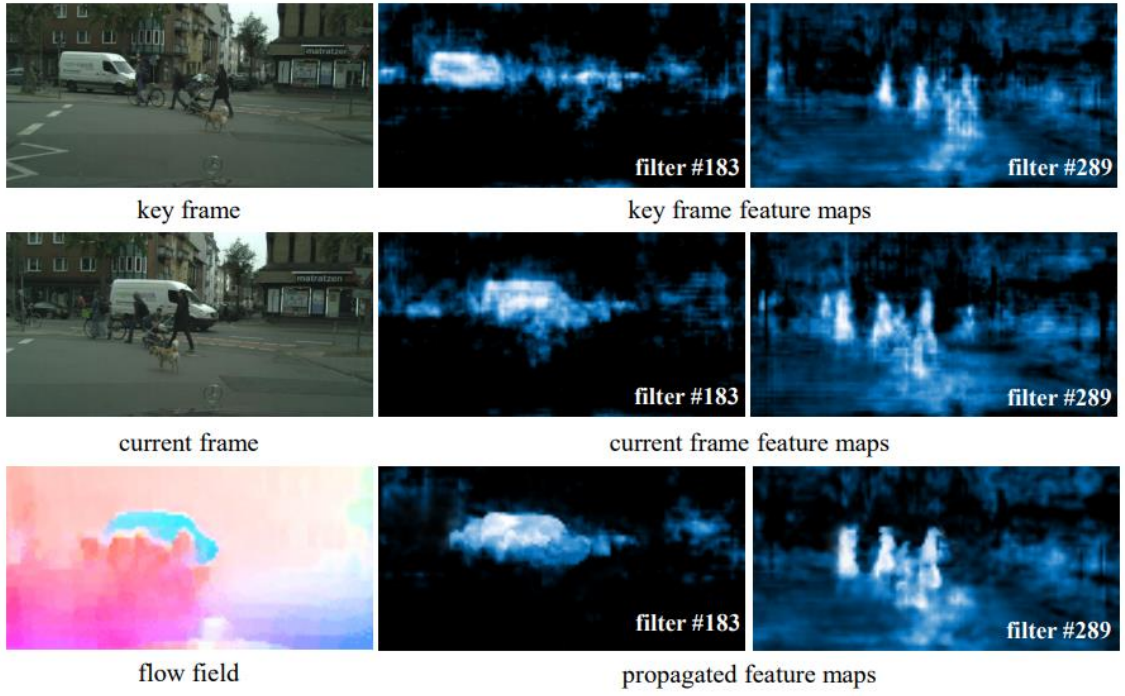
: Deep Feature Flow for Video Recognition

2017 CVPR

Introduction

"Fast and Accurate Video Recognition"

For Autonomous Driving, Video Surveillance



- Flow estimation
  : Apply image recognition network on sparse key frames
- Feature propagation
  Propagates the deep feature maps from key frames to other frames via a flow field

⇒ **fast, accurate, general, end-to-end framework**

Deep Feature Flow

Deep Feature Flow Inference

- Bilinear-interpolation
  : $f_i^c(p) = \sum_q G(q, p + \delta p) f_k^c(q)$
- $G(q, p + \delta p) = g(q_x, p_x + \delta p_x) \cdot g(q_y, p_y + \delta p_y)$
  (where $g(a, b) = \max(0, 1 - |a - b|)$)
- Scale field : $S_{i \to k} = S(I_k, I_i)$
- Feature propagation
  : $f_i = \mathcal{W}(f_k, M_{i \to k}, S_{i \to k})$

Algorithm 1 Deep feature flow inference algorithm for video recognition.

input: video frames $\{\mathbf{I}_i\}$
$k = 0$; ▷ initialize key frame
$\mathbf{f}_0 = \mathcal{N}_{feat}(\mathbf{I}_0)$
$\mathbf{y}_0 = \mathcal{N}_{task}(\mathbf{f}_0)$
for $i = 1$ **to** ∞ **do**
  if $is\_key\_frame(i)$ **then** ▷ key frame scheduler
    $k = i$ ▷ update the key frame
    $\mathbf{f}_k = \mathcal{N}_{feat}(\mathbf{I}_k)$
    $\mathbf{y}_k = \mathcal{N}_{task}(\mathbf{f}_k)$
  else ▷ use feature flow
    $\mathbf{f}_i = \mathcal{W}(\mathbf{f}_k, \mathcal{F}(\mathbf{I}_k, \mathbf{I}_i), \mathcal{S}(\mathbf{I}_k, \mathbf{I}_i))$ ▷ propagation
    $\mathbf{y}_i = \mathcal{N}_{task}(\mathbf{f}_i)$
  end if
end for
output: recognition results $\{\mathbf{y}_i\}$

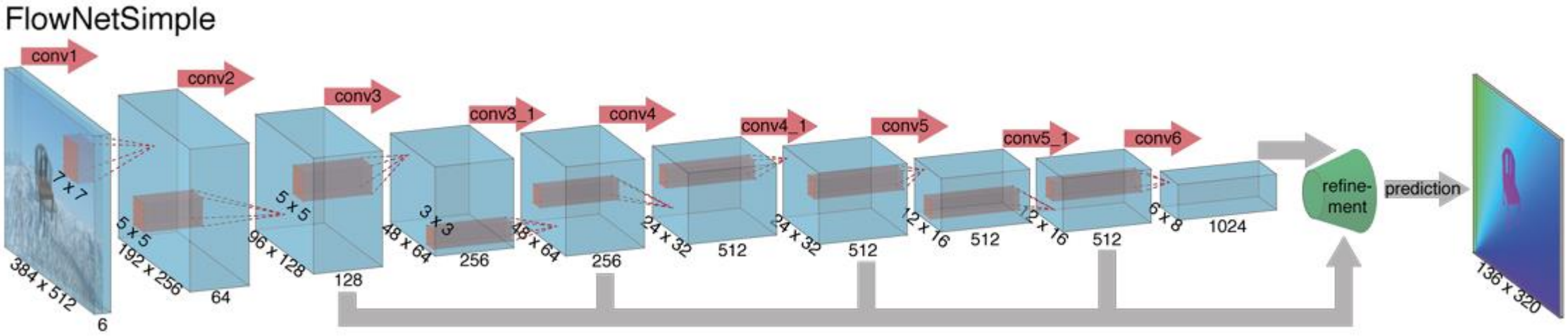
k	key frame index
i	current frame index
r	per-frame computation cost ratio, Eq. (5)
l	key frame duration length
s	overall speedup ratio, Eq. (7)
$\mathbf{I}_i, \mathbf{I}_k$	video frames
$\mathbf{y}_i, \mathbf{y}_k$	recognition results
$\mathbf{f}_k$	convolutional feature maps on key frame
$\mathbf{f}_i$	propagated feature maps on current frame
$\mathbf{M}_{i \to k}$	2D flow field
$\mathbf{p}, \mathbf{q}$	2D location
$\mathbf{S}_{i \to k}$	scale field
$\mathcal{N}$	image recognition network
$\mathcal{N}_{feat}$	sub-network for feature extraction
$\mathcal{N}_{task}$	sub-network for recognition result
$\mathcal{F}$	flow estimation function
$\mathcal{W}$	feature propagation function, Eq. (3)

Table 1. Notations.

Feature Propagation

$$\frac{\partial f_i^c(p)}{\partial M_{i \to k}(p)} = S_{i \to k}^c(p) \sum_q \frac{\partial G(q, p + \delta p)}{\partial \delta p} f_k^c(q)$$ → parameter free & fully differentiable

FlowNet



① FlowNet Half
: reduce convolutional kernels in each layer $\frac{1}{2}$ and complexity to $\frac{1}{4}$

② FlowNet Inception
: Adopt Inception structure and reduces complexity to $\frac{1}{8}$

Pre-trained on synthetic flying chair dataset

Structure

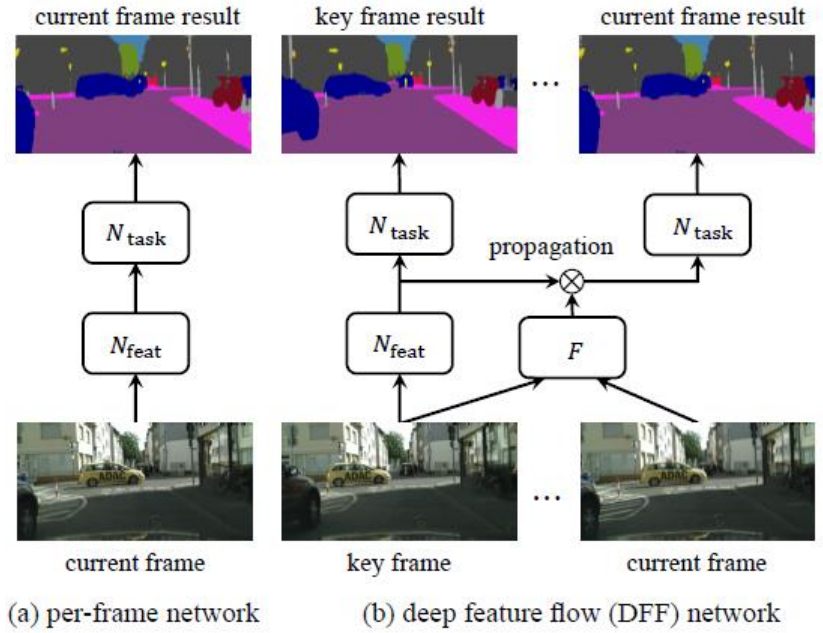
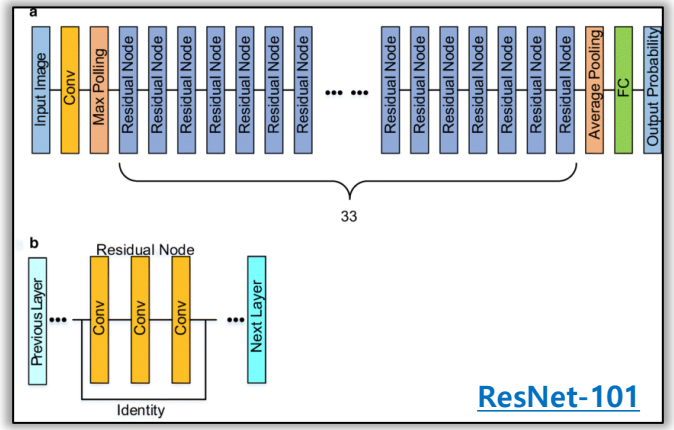
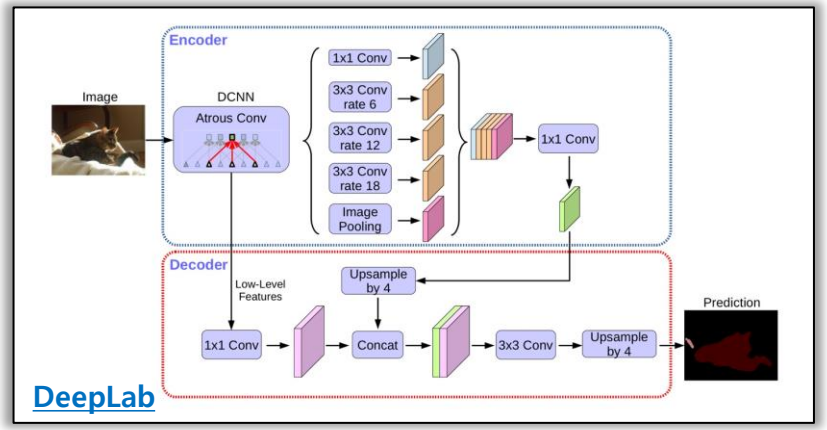


Figure 2. Illustration of video recognition using per-frame network evaluation (a) and the proposed deep feature flow (b).

layer	type	stride	# output	Inception/Reduction			
				#1x1	#1x1-#3x3	#1x1-#3x3-#3x3	#pool
conv1	7x7 conv	2	32				
pool1	3x3 max pool	2	32				
conv2	Inception		64		24-32	24-32-32	
conv3_1	3x3 conv	2	128				
conv3_2	Inception		128	48	32-64	8-16-16	
conv3_3	Inception		128	48	48-64	12-16-16	
conv4_1	Reduction	2	256	32	112-128	28-32-32	64
conv4_2	Inception		256	96	112-128	28-32-32	
conv5_1	Reduction	2	384	48	96-192	36-48-48	96
conv5_2	Inception		384	144	96-192	36-48-48	
conv6_1	Reduction	2	512	64	192-256	48-64-64	128
conv6_2	Inception		512	192	192-256	48-64-64	

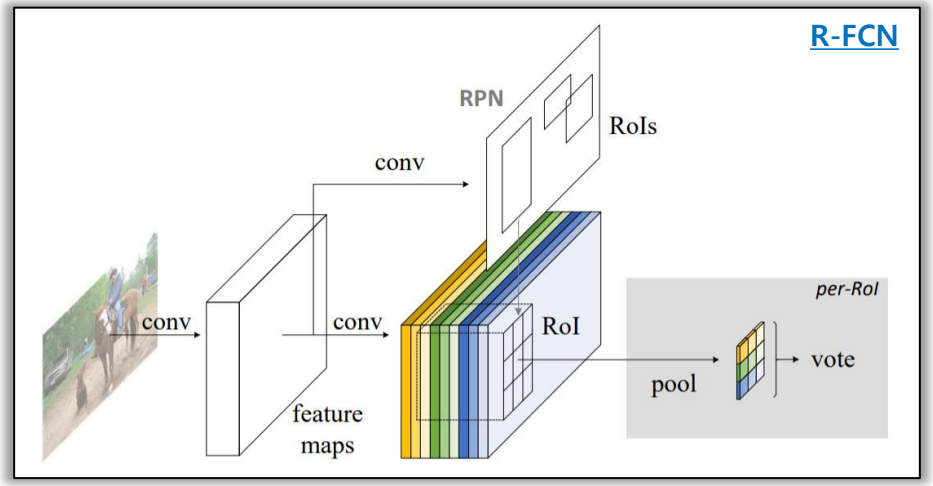




ResNet-50 or ResNet-101 pre-trained for ImageNet

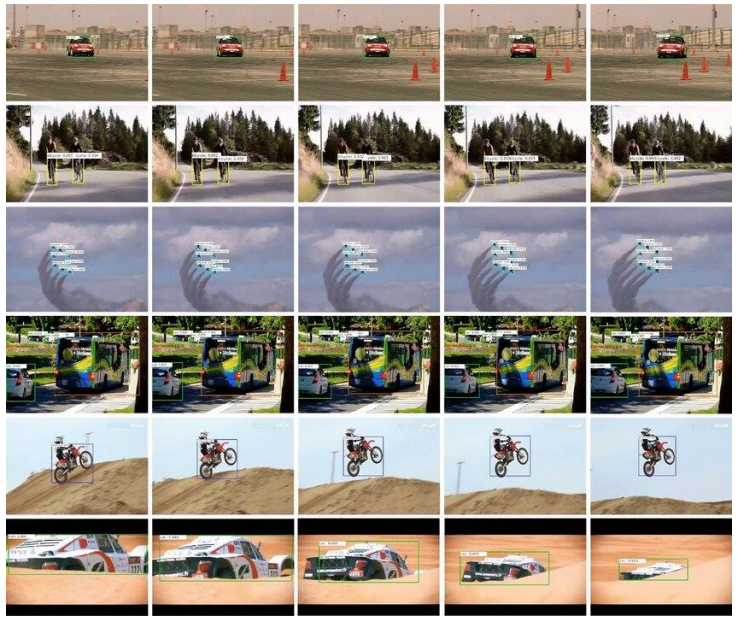
- Semantic Segmentation : DeepLab
- Object Detection : R-FCN

	FlowNet	FlowNet Half	FlowNet Inception
ResNet-50	9.20	33.56	68.97
ResNet-101	12.71	46.30	95.24



Experiment

Datasets



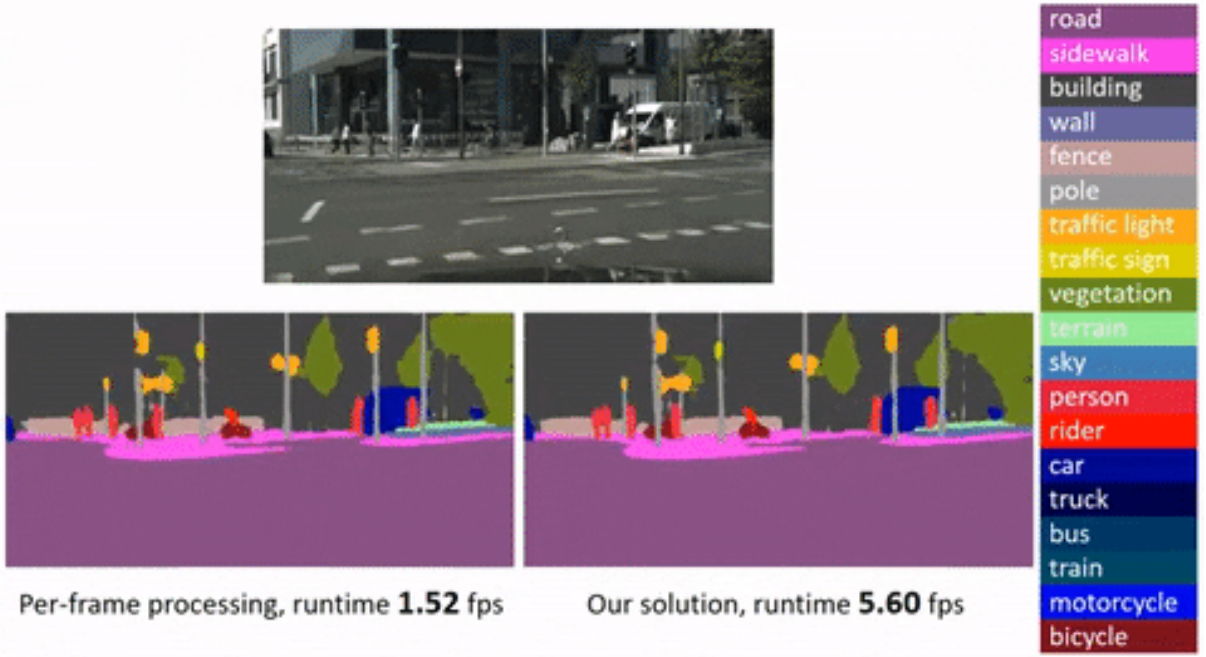
- Cityscapes
  - For urban scene understanding and autonomous driving
  - 2975,500 dataset, and 1525 snippets respectively
  - Feature network : 1024 pixels / Flow network : 512 pixels

- ImageNet VID
  - For object detection in Videos
  - 3,862,555 dataset, and 937 fully-annotated video snippets respectively
  - Feature network : 600 pixels / Flow network : 300 pixels

Result

Per-frame processing, runtime **4.05** fps

Our solution, runtime **20.25** fps



Per-frame processing, runtime **1.52** fps

Our solution, runtime **5.60** fps

Reference

- Kaiming He, Georgia Gkioxari, Piotr Dollar, Ross Girschick.「**Mask R-CNN**」, 2017
- Jonathan Long, Evan Shelhamer, Trevor Darrell.「**Fully Convolutional Networks for Semantic Segmentation**」, 2014
- Alexander Kirillov, Kaiming He, Ross Girschick, Carsten Rother, Piotr Dollar.「**Panoptic Segmentation**」, 2018
- Jifeng Dai, Kaiming He, Jain Sun.「**Instance-aware Semantic Segmentation via Multi-task Network Cascades**」**,** 2016
- Jifeng Dai, Yi Li, Haozhi Qi, Xiangyang Ji, Yichen Wei.「**Fully Convolutional Instance-aware Semantic Segmentation**」**,** 2017
- Matthew D.Zeiler, Rob Furgus.「**Visualizing and Understanding Convolutional Networks**」**,** 2013
- Xizhou Zhu, Yuwen Xiong, Jifeng Dai, Lu Yuan, Yichen Wei.「**Deep Feature Flow for Video Recognition**」, 2017
- Philipp Fischer, Alexey Dosovitskiy, Eddy Ilg, Philip Häusser, Caner Hazirbaş, Vladimir Golkov, Patrick van der Smagt, Daniel Cremers, Thomas Brox.「**FlowNet: Learning Optical Flow with Convolutional Networks**」, 2015
- Hyeonwoo Noh, Seunghoon Hong, Bohyung Han.「**Learning Deconvolution Network for Semantic Segmentation**」, 2015
- Stanford. CS231n
- https://www.kaggle.com/datasets/faizalkarim/flood-area-segmentation
- https://image-net.org/challenges/LSVRC/2017/

THANK YOU