You Only Segment Once: Towards Real-Time Panoptic Segmentation
Supplementary Material

A. Qualitative Results for Panoptic Segmentation

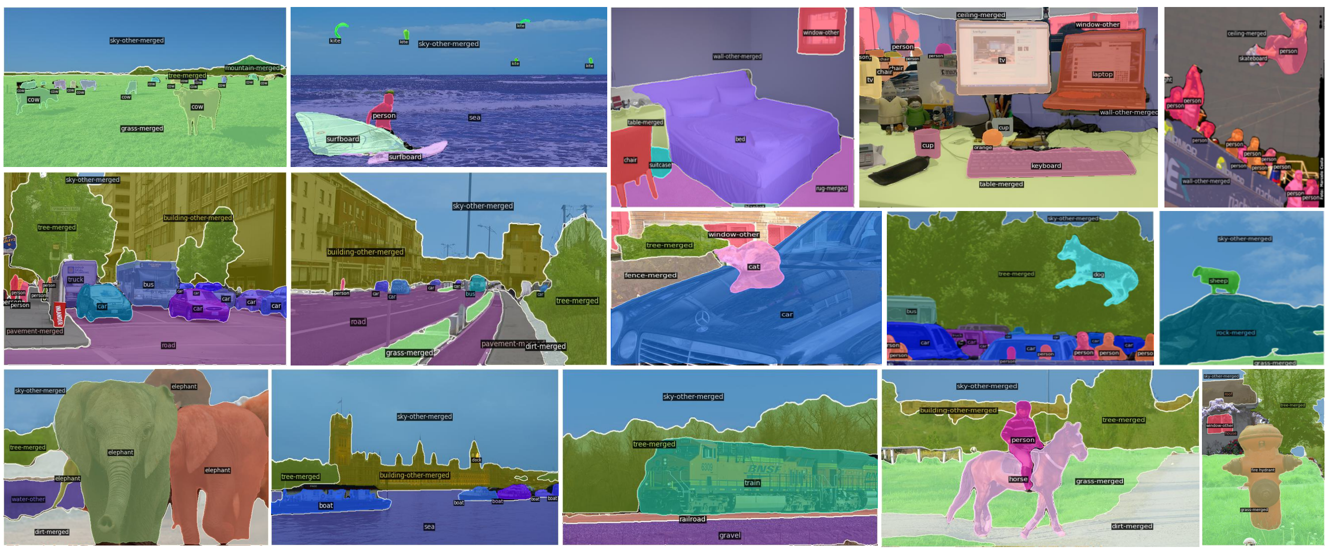

Figure 6. Panoptic segmentation on the **COCO** *validation* set.


Figure 7. Panoptic segmentation on the **Cityscapes** *validation* set.

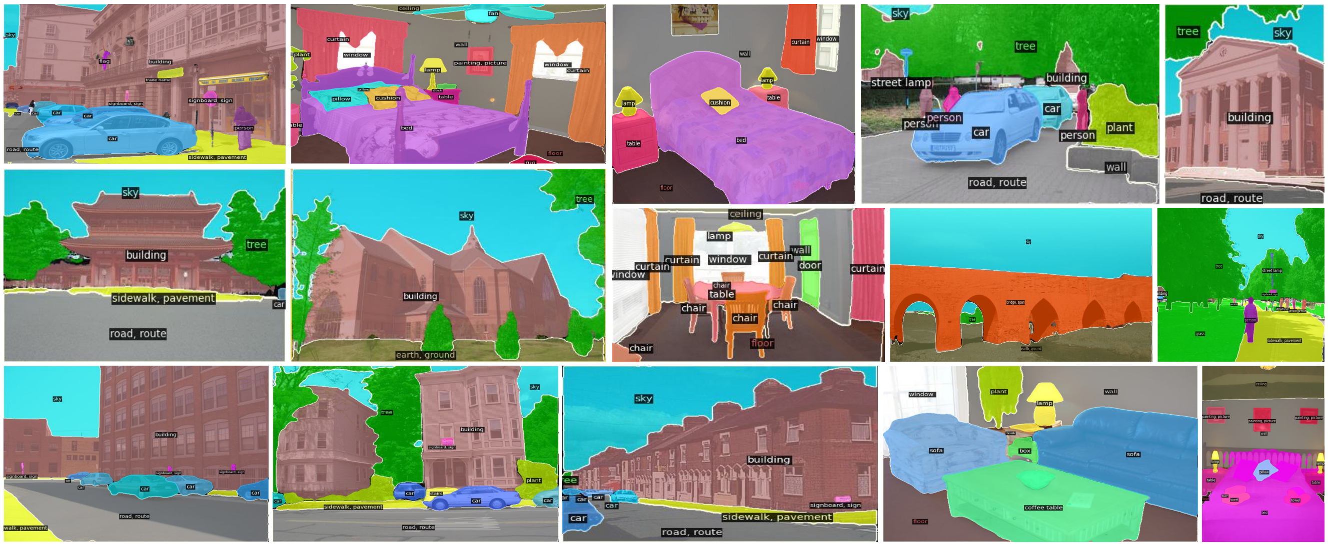

Figure 8. Panoptic segmentation on the **ADE20K** *validation* set.

Figure 9. Panoptic segmentation on the **Mapillary Vistas** *validation* set.

B. Quantitive Results on Real-Time Instance Segmentation

Solving the instance segmentation task efficiently is one of the keys to achieving real-time panoptic segmentation. Therefore, we study the performance of YOSO for real-time instance segmentation in Tab. 11. Note that the model is not specifically trained for instance segmentation. We show the results from the model trained on the COCO training set for panoptic segmentation. From the results on Tab. 11, we find that YOSO also achieves competitive performance for instance segmentation on the COCO validation set. When scaling the input images to 550, YOSO achieves 38.7 FPS and 34.7 mAP. The speed is only 5.9 FPS lower than the current state-of-the-art model, *i.e.*, SparseInst, while the mAP is 0.3 points higher. Specifically, the performance of YOSO on large objects, *i.e.*, AP^l, is better than all the state-of-the-art models. For example, when scaling the input images to 448, YOSO still can achieve 57.6 AP^l for large objects, which is approximately 2.2 points higher than the performance of SOLOv2. The result suggests that YOSO is good at segmenting large objects.

Method	Backbone	Scale	AP	AP_{50}	AP_{75}	AP^s	AP^m	AP^l	GPU	FPS
YOLACT [1]	DarkNet-53	550	28.9	46.9	30.3	9.8	30.9	47.3	2080Ti	45.9
YOLACT++ [2]	ResNet-50	550	33.7	52.7	35.5	11.9	36.6	54.6	2080Ti	40.8
BlendMask [3]	ResNet-50	550	34.5	54.7	36.5	14.4	37.7	52.1	2080Ti	35.6
SOLOv2 [6]	ResNet-50	448	33.7	53.3	35.6	11.3	36.9	55.4	2080Ti	39.6
OrienMask [5]	DarkNet-53	544	34.5	56.0	35.8	16.8	38.5	49.1	2080Ti	41.9
SparseInst [4]	ResNet-50	608	34.4	55.2	36.1	14.2	36.8	51.9	2080Ti	44.6
YOLO, *ours*	ResNet-50	448	33.0	52.8	34.6	11.3	35.4	57.6	2080Ti	46.1
YOSO, *ours*	ResNet-50	550	34.7	55.0	36.4	13.2	37.6	58.6	2080Ti	38.7
YOSO, *ours*	ResNet-50	608	35.6	56.3	37.5	14.4	39.0	59.2	2080Ti	33.5

Table 11. Real-time instance segmentation results on the **COCO** *validation* set.

C. Codes

In this section, we provide the codes for the proposed components in YOSO as well as for the computation of FLOPs and GPU latency. Specifically, in Code. 1, we provide the implementation for IFA and CFA, which demonstrates: 1) the outputs of IFA and CFA are exactly the same; 2) the number of FLOPs for CFA is less than that for IFA; 3) CFA runs faster than CFA in terms of GPU latency. Similarly, in Code. 2, we provide the implementation for DCA, DSCA, DDCA, and DPCA. As the FLOPs counter does not compute the FLOPs for 'nn.functional.conv1d', we add the number of this operation in L171-172 and L178-181.

Listing 1. Pytorch code for computing FLOPs and GPU latency of different aggregation blocks.

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from fvcore.nn import FlopCountAnalysis


class IFAModule(nn.Module):
   def __init__(self, in_channels=128, out_channels=[1024, 512, 256, 128]):
      super().__init__()
      self.weights = nn.Parameter(torch.rand((in_channels, sum(out_channels), 1, 1)))

   def forward(self, p5, p4, p3, p2):
      x5 = F.interpolate(p5, scale_factor=8, align_corners=False, mode='bilinear')
      x4 = F.interpolate(p4, scale_factor=4, align_corners=False, mode='bilinear')
      x3 = F.interpolate(p3, scale_factor=2, align_corners=False, mode='bilinear')
      x2 = p2
      x_fuse = torch.concat([x5, x4, x3, x2], dim=1)
      # x_fuse: [1, 1920, 256, 256]
      output = F.conv2d(x_fuse, self.weights)
      return output


class CFAModule(nn.Module):
   def __init__(self, in_channels=128, out_channels=[1024, 512, 256, 128]):
      super().__init__()
      self.weights = nn.Parameter(torch.rand((in_channels, sum(out_channels), 1, 1)))
      self.out_channels = out_channels

   def forward(self, p5, p4, p3, p2):
      x5 = F.interpolate(F.conv2d(p5, self.weights[:, :self.out_channels[0]]),
          scale_factor=8, align_corners=False, mode='bilinear')
      x4 = F.interpolate(F.conv2d(p4, self.weights[:,
          sum(self.out_channels[:1]):sum(self.out_channels[:2])]), scale_factor=4,
          align_corners=False, mode='bilinear')
```

```python
        x3 = F.interpolate(F.conv2d(p3, self.weights[:,
            sum(self.out_channels[:2]):sum(self.out_channels[:3])]), scale_factor=2,
            align_corners=False, mode='bilinear')
        x2 = F.conv2d(p2, self.weights[:, sum(self.out_channels[:3]):])
        output = x5 + x4 + x3 + x2
        return output


class CFAModuleForFlops(nn.Module):
    def __init__(self, in_channels=128, out_channels=[1024, 512, 256, 128]):
        super().__init__()
        self.weights = nn.Parameter(torch.rand((in_channels, sum(out_channels), 1, 1)))
        self.conv = nn.Conv3d(4, 1, 1, bias=False)
        nn.init.constant_(self.conv.weight, 1)
        self.out_channels = out_channels

    def forward(self, p5, p4, p3, p2):
        x5 = F.interpolate(F.conv2d(p5, self.weights[:, :self.out_channels[0]]),
            scale_factor=8, align_corners=False, mode='bilinear')
        x4 = F.interpolate(F.conv2d(p4, self.weights[:,
            sum(self.out_channels[:1]):sum(self.out_channels[:2])]), scale_factor=4,
            align_corners=False, mode='bilinear')
        x3 = F.interpolate(F.conv2d(p3, self.weights[:,
            sum(self.out_channels[:2]):sum(self.out_channels[:3])]), scale_factor=2,
            align_corners=False, mode='bilinear')
        x2 = F.conv2d(p2, self.weights[:, sum(self.out_channels[:3]):])
        concat = torch.concat([x5.unsqueeze(0), x4.unsqueeze(0), x3.unsqueeze(0),
            x2.unsqueeze(0)], dim=0)
        concat = concat.permute(1, 0, 2, 3, 4)

        # To compute the number of FLOPs for accumulation operation
        output = self.conv(concat)
        # output = x5 + x4 + x3 + x2
        # output = torch.add([x5, x4, x3, x2])
        return output


# ------ FLOPs ------
inputs = (
    torch.rand((1, 1024, 32, 32)),
    torch.rand((1, 512, 64, 64)),
    torch.rand((1, 256, 128, 128)),
    torch.rand((1, 128, 256, 256)),
)

IFA = IFAModule()
CFA = CFAModuleForFlops()

print()
flops = FlopCountAnalysis(IFA, inputs)
print("IFA flops counter: ")
print(flops.total())
print(flops.by_operator())

print()
flops = FlopCountAnalysis(CFA, inputs)
print("CFA flops counter: ")
print(flops.total())
```

```
print(flops.by_operator())

# ------ GPU Latency ------
inputs = (
    torch.rand((1, 1024, 32, 32)).cuda(),
    torch.rand((1, 512, 64, 64)).cuda(),
    torch.rand((1, 256, 128, 128)).cuda(),
    torch.rand((1, 128, 256, 256)).cuda(),
)

IFA = IFAModule().cuda()
CFA = CFAModule().cuda()
IFA.weights.data = CFA.weights.data

# warm up
ifa_output = IFA(inputs[0], inputs[1], inputs[2], inputs[3])
cfa_output = CFA(inputs[0], inputs[1], inputs[2], inputs[3])
print("check the same output: ", (cfa_output.sum() - ifa_output.sum()))

with torch.autograd.profiler.profile(enabled=True, use_cuda=True, record_shapes=False)
     as prof:
    IFA(inputs[0], inputs[1], inputs[2], inputs[3])
# NOTE: some columns were removed for brevity
print(prof.key_averages().table(sort_by="self_cpu_time_total"))

with torch.autograd.profiler.profile(enabled=True, use_cuda=True, record_shapes=False)
     as prof:
    CFA(inputs[0], inputs[1], inputs[2], inputs[3])
# NOTE: some columns were removed for brevity
print(prof.key_averages().table(sort_by="self_cpu_time_total"))
```

Listing 2. Pytorch code for computing FLOPs and GPU latency of different attention types.

```
import torch
from torch import nn
import torch.nn.functional as F
from timm.models.layers import trunc_normal_

from fvcore.nn import FlopCountAnalysis
from fvcore.nn import ActivationCountAnalysis
from ptflops import get_model_complexity_info

HIDDEN_DIM = 256
NUM_PROPOSALS = 100
CONV_KERNEL_SIZE_1D = 3


class MultiHeadCrossAtten(nn.Module):
    def __init__(self):
        super(MultiHeadCrossAtten, self).__init__()
        self.hidden_dim = HIDDEN_DIM
        self.num_proposals = NUM_PROPOSALS
        self.conv_kernel_size_1d = CONV_KERNEL_SIZE_1D

        self.atten = nn.MultiheadAttention(embed_dim=self.hidden_dim * 1**2, num_heads=8,
            dropout=0.0)
        self.f_norm = nn.LayerNorm(self.hidden_dim)
```

```python

    def forward(self, query, value):
        query = query.permute(1, 0, 2)
        value = value.permute(1, 0, 2)

        out = self.atten(query, value, value)[0]
        out = out.permute(1, 0, 2)
        out = self.f_norm(out)
        return out


class DyConvAtten(nn.Module):
    def __init__(self):
        super(DyConvAtten, self).__init__()
        self.hidden_dim = HIDDEN_DIM
        self.num_proposals = NUM_PROPOSALS
        self.conv_kernel_size_1d = CONV_KERNEL_SIZE_1D

        self.f_linear = nn.Linear(self.hidden_dim, self.num_proposals * self.conv_kernel_size_1d)
        self.f_norm = nn.LayerNorm(self.hidden_dim)

    def forward(self, f, k):
        # f: [B, N, C]
        # k: [B, N, C * K * K]
        B = f.shape[0]
        weight = self.f_linear(f)
        weight = weight.view(B, self.num_proposals, self.num_proposals, self.conv_kernel_size_1d)
        res = []
        for i in range(B):
            # input: [1, N, C * K * K]
            # weight: [N, N, convK]
            # output: [1, N, C * K * K]
            out = F.conv1d(input=k.unsqueeze(1)[i], weight=weight[i], padding='same')
            res.append(out)
        # [B, N, C * K * K]
        f_tmp = torch.cat(res, dim=0)
        f_tmp = self.f_norm(f_tmp)
        return f_tmp


class DySepConvAtten(nn.Module):
    def __init__(self):
        super(DySepConvAtten, self).__init__()
        self.hidden_dim = HIDDEN_DIM
        self.num_proposals = NUM_PROPOSALS
        self.kernel_size = CONV_KERNEL_SIZE_1D

        self.weight_linear = nn.Linear(self.hidden_dim, self.num_proposals + self.kernel_size)
        self.norm = nn.LayerNorm(self.hidden_dim)

    def forward(self, query, value):
        assert query.shape == value.shape
        B, N, C = query.shape
        dy_conv_weight = self.weight_linear(query)
        res = []
```

```
        value = value.unsqueeze(1)
        for i in range(B):
            # input: [1, N, C]
            # weight: [N, 1, K]
            # output: [1, N, C]
            out = F.relu(F.conv1d(input=value[i], weight=dy_conv_weight[i, :,
                :self.kernel_size].view(self.num_proposals,1,self.kernel_size), groups=N,
                padding="same"))
            # input: [1, N, C]
            # weight: [N, N, 1]
            # output: [1, N, C]
            out = F.conv1d(input=out, weight=dy_conv_weight[i, :,
                self.kernel_size:].view(self.num_proposals,self.num_proposals,1),
                padding='same')

            res.append(out)
        point_out = torch.cat(res, dim=0)
        point_out = self.norm(point_out)
        return point_out


class DyDepthwiseConvAtten(nn.Module):
    def __init__(self):
        super(DyDepthwiseConvAtten, self).__init__()
        self.hidden_dim = HIDDEN_DIM
        self.num_proposals = NUM_PROPOSALS
        self.kernel_size = CONV_KERNEL_SIZE_1D

        self.weight_linear = nn.Linear(self.hidden_dim, self.kernel_size)
        self.norm = nn.LayerNorm(self.hidden_dim)

    def forward(self, query, value):
        assert query.shape == value.shape
        B, N, C = query.shape
        dy_conv_weight =
            self.weight_linear(query).view(B,self.num_proposals,1,self.kernel_size)

        res = []
        value = value.unsqueeze(1)
        for i in range(B):
            # input: [1, N, C]
            # weight: [N, 1, K]
            # output: [1, N, C]
            out = F.conv1d(input=value[i], weight=dy_conv_weight[i], groups=N,
                padding="same")
            res.append(out)
        point_out = torch.cat(res, dim=0)
        point_out = self.norm(point_out)
        return point_out


class DyPointwiseConvAtten(nn.Module):
    def __init__(self):
        super(DyPointwiseConvAtten, self).__init__()
        self.hidden_dim = HIDDEN_DIM
        self.num_proposals = NUM_PROPOSALS
        self.kernel_size = CONV_KERNEL_SIZE_1D

```

```
        self.weight_linear = nn.Linear(self.hidden_dim, self.num_proposals)
        self.norm = nn.LayerNorm(self.hidden_dim)

    def forward(self, query, value):
        assert query.shape == value.shape
        B, N, C = query.shape

        dy_conv_weight =
            self.weight_linear(query).view(B,self.num_proposals,self.num_proposals,1)

        res = []
        value = value.unsqueeze(1)
        for i in range(B):
            # input: [1, N, C]
            # weight: [N, N, 1]
            # output: [1, N, C]
            out = F.conv1d(input=value[i], weight=dy_conv_weight[i], padding='same')

            res.append(out)
        point_out = torch.cat(res, dim=0)
        point_out = self.norm(point_out)
        return point_out


# ------ FLOPs ------
q = torch.rand((1, NUM_PROPOSALS, HIDDEN_DIM))
v = torch.rand((1, NUM_PROPOSALS, HIDDEN_DIM))

MHCA = MultiHeadCrossAtten()
DCA = DyConvAtten()
DSCA = DySepConvAtten()
DDCA = DyDepthwiseConvAtten()
DPCA = DyPointwiseConvAtten()

flops = FlopCountAnalysis(MHCA, (q, v))
print("MHCA flops counter: ")
print(flops.total())
print(flops.by_operator())

flops = FlopCountAnalysis(DCA, (q, v))
print("DCA flops counter: ")
conv = nn.Conv1d(in_channels=NUM_PROPOSALS, out_channels=NUM_PROPOSALS,
    kernel_size=CONV_KERNEL_SIZE_1D, bias=False, padding='same')
macs, _ = get_model_complexity_info(conv, (NUM_PROPOSALS, HIDDEN_DIM),
    as_strings=False, print_per_layer_stat=False, verbose=True)
print(flops.total() + macs)
print(flops.by_operator(), "conv: ", macs)

flops = FlopCountAnalysis(DSCA, (q, v))
print("DSCA flops counter: ")
depthwise_conv = nn.Conv1d(in_channels=NUM_PROPOSALS, out_channels=NUM_PROPOSALS,
    kernel_size=CONV_KERNEL_SIZE_1D, bias=False, groups=NUM_PROPOSALS, padding='same')
macs_depthwise, _ = get_model_complexity_info(depthwise_conv, (NUM_PROPOSALS,
    HIDDEN_DIM), as_strings=False, print_per_layer_stat=False, verbose=True)
pointwise_conv = nn.Conv1d(in_channels=NUM_PROPOSALS, out_channels=NUM_PROPOSALS,
    kernel_size=1, bias=False, padding='same')
macs_pointwise, _ = get_model_complexity_info(pointwise_conv, (NUM_PROPOSALS,
    HIDDEN_DIM), as_strings=False, print_per_layer_stat=False, verbose=True)
```

```
print(flops.total() + macs_depthwise + macs_pointwise)
print(flops.by_operator(), "depthwise: ", macs_depthwise, "pointwise: ", macs_pointwise)

flops = FlopCountAnalysis(DDCA, (q, v))
print("DDCA flops counter: ")
print(flops.total() + macs_depthwise)
print(flops.by_operator(), "depthwise: ", macs_depthwise,)

flops = FlopCountAnalysis(DPCA, (q, v))
print("DPCA flops counter: ")
print(flops.total() + macs_pointwise)
print(flops.by_operator(), "pointwise: ", macs_pointwise)


# ------ GPU Latency ------
q = torch.rand((1, NUM_PROPOSALS, HIDDEN_DIM), requires_grad=False).cuda()
v = torch.rand((1, NUM_PROPOSALS, HIDDEN_DIM), requires_grad=False).cuda()

MHCA = MultiHeadCrossAtten().cuda()
DCA = DyConvAtten().cuda()
DSCA = DySepConvAtten().cuda()
DDCA = DyDepthwiseConvAtten().cuda()
DPCA = DyPointwiseConvAtten().cuda()

# warm up
o = MHCA(q, v)
o = DCA(q, v)
o = DSCA(q, v)
o = DDCA(q, v)
o = DPCA(q, v)

with torch.autograd.profiler.profile(enabled=True, use_cuda=True, record_shapes=False) as prof:
    o = MHCA(q, v)
# NOTE: some columns were removed for brevity
print(prof.key_averages().table(sort_by="self_cpu_time_total"))

with torch.autograd.profiler.profile(enabled=True, use_cuda=True, record_shapes=False) as prof:
    o = DCA(q, v)
# NOTE: some columns were removed for brevity
print(prof.key_averages().table(sort_by="self_cpu_time_total"))

with torch.autograd.profiler.profile(enabled=True, use_cuda=True, record_shapes=False) as prof:
    o = DSCA(q, v)
# NOTE: some columns were removed for brevity
print(prof.key_averages().table(sort_by="self_cpu_time_total"))

with torch.autograd.profiler.profile(enabled=True, use_cuda=True, record_shapes=False) as prof:
    o = DDCA(q, v)
# NOTE: some columns were removed for brevity
print(prof.key_averages().table(sort_by="self_cpu_time_total"))

with torch.autograd.profiler.profile(enabled=True, use_cuda=True, record_shapes=False) as prof:
    o = DPCA(q, v)
```

```python
# NOTE: some columns were removed for brevity
print(prof.key_averages().table(sort_by="self_cpu_time_total"))
```

References

[1] Daniel Bolya, Chong Zhou, Fanyi Xiao, and Yong Jae Lee. Yolact: Real-time instance segmentation. In *Proceedings of the IEEE International Conference on Computer Vision*, 2019. 3

[2] Daniel Bolya, Chong Zhou, Fanyi Xiao, and Yong Jae Lee. Yolact++: Better real-time instance segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2020. 3

[3] Hao Chen, Kunyang Sun, Zhi Tian, Chunhua Shen, Yongming Huang, and Youliang Yan. Blendmask: Top-down meets bottom-up for instance segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2020. 3

[4] Tianheng Cheng, Xinggang Wang, Shaoyu Chen, Wenqiang Zhang, Qian Zhang, Chang Huang, Zhaoxiang Zhang, and Wenyu Liu. Sparse instance activation for real-time instance segmentation. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2022. 3

[5] Wentao Du, Zhiyu Xiang, Shuya Chen, Chengyu Qiao, Yiman Chen, and Tingming Bai. Real-time instance segmentation with discriminative orientation maps. In *Proceedings of the IEEE International Conference on Computer Vision*, 2021. 3

[6] Xinlong Wang, Rufeng Zhang, Tao Kong, Lei Li, and Chunhua Shen. Solov2: Dynamic and fast instance segmentation. *Advances in Neural Information Processing Systems*, 2020. 3