

# **Cascaded Pyramid Network for Multi-Person Pose Estimation (2017)**

Seho Kim

<https://arxiv.org/pdf/1711.07319.pdf>

# Introduction

- The problem of multi-person pose estimation has been greatly improved by the involvement of deep convolutional neural networks. (ex. PAFs)
- Mask-RCNN → Bbox → warps feature maps → keypoints
- Challenging cases (such as occluded keypoints, invisible keypoints and crowded background)

# Introduction

- CPN (Cascaded Pyramid Network)
  - Two stages: GlobalNet and RefineNet
  - GlobalNet: good feature representation(FPN)
  - RefineNet: explicitly address the ‘hard’ joints  
(online hard keypoints mining loss)
  - Top-down pipeline
  - SOTA(2017)
    - ; 73.0 AP in test-dev dataset
    - 72.1 AP in test challenge dataset

# Approach

- Human Detector
  - SOTA object detector algorithms based on FPN
  - ROIALign(Mask RCNN); to replace the ROI Pooling in FPN

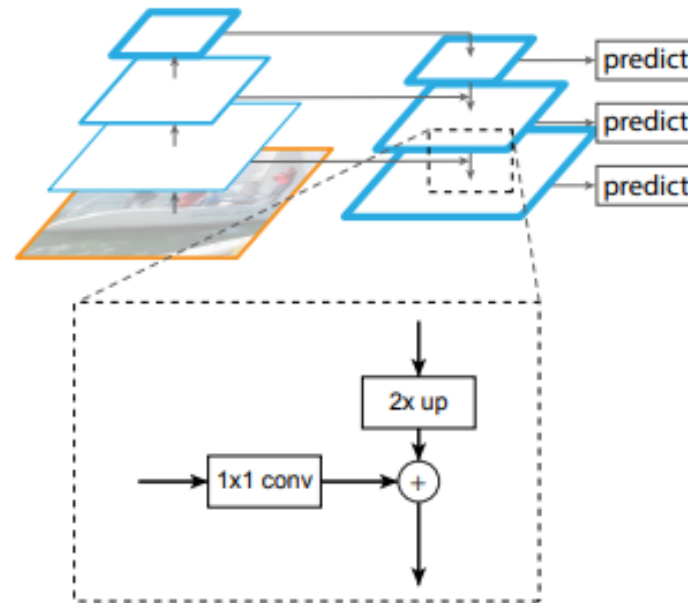


Figure 3. A building block illustrating the lateral connection and the top-down pathway, merged by addition.

# Approach

- Cascaded Pyramid Network (CPN)
  - Stacked hourglass
  - Stacking two hourglasses
  - Utilizes a ResNet network
  - Two sub-networks: GlobalNet and RefineNet

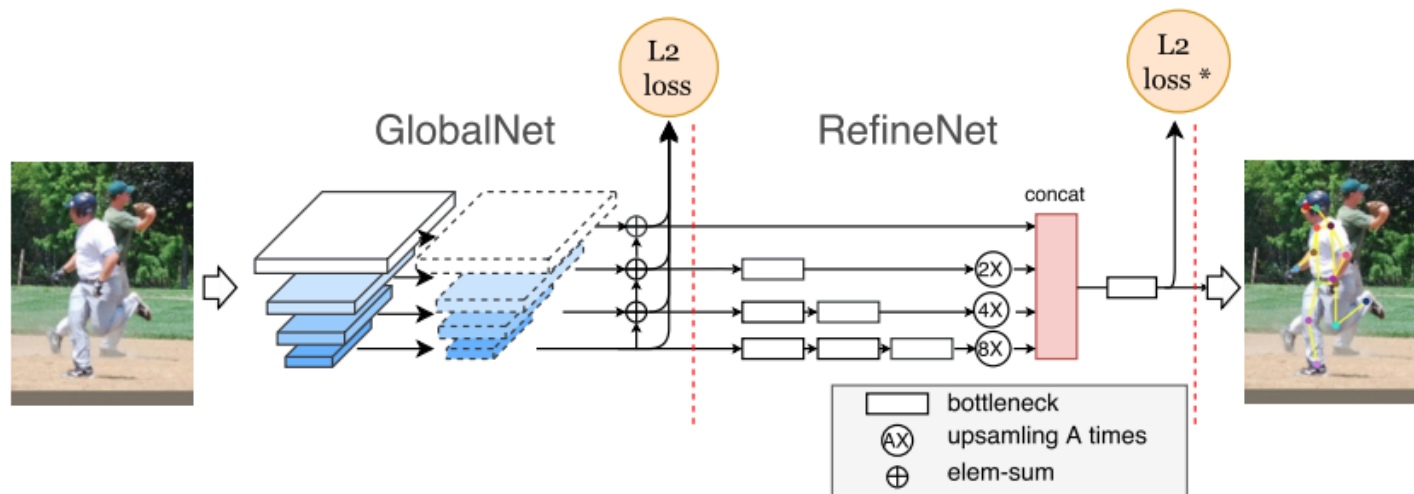


Figure 1. Cascaded Pyramid Network. “L2 loss\*” means L2 loss with online hard keypoints mining.

# Approach

- GlobalNet
  - Based on the ResNet backbone
  - 3x3 convolution filters
    - ; conv2, conv3 high spatial resolution for localization but low semantic information
    - ; conv4, conv5 more semantic information but low spatial resolution

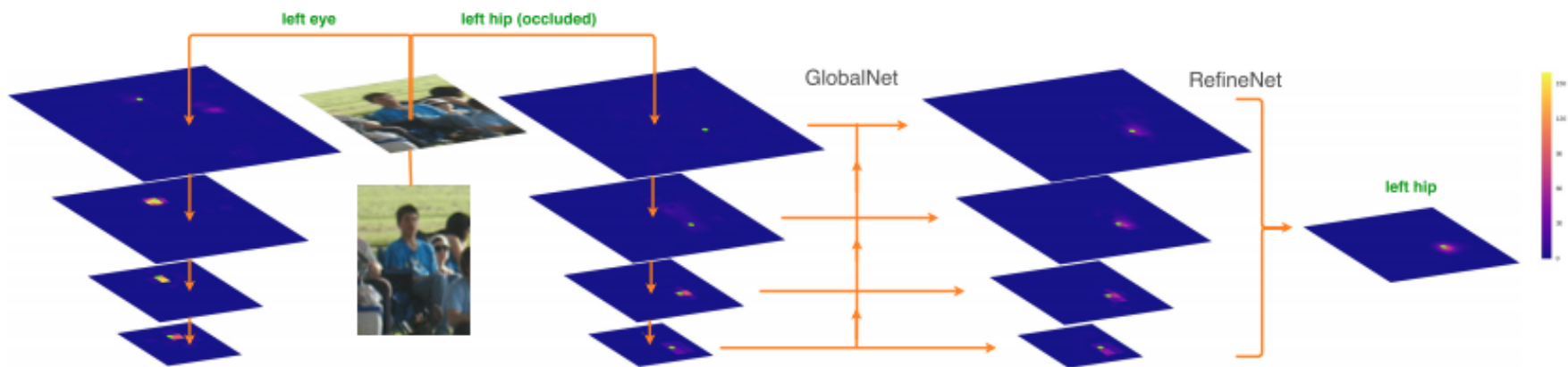


Figure 2. Output heatmaps from different features. The green dots means the groundtruth location of keypoints.

# Approach

- GlobalNet
  - U-shape structure + FPN = feature pyramid structure
  - 1x1 convolutional kernel
    - element-wise sum (in upsampling)

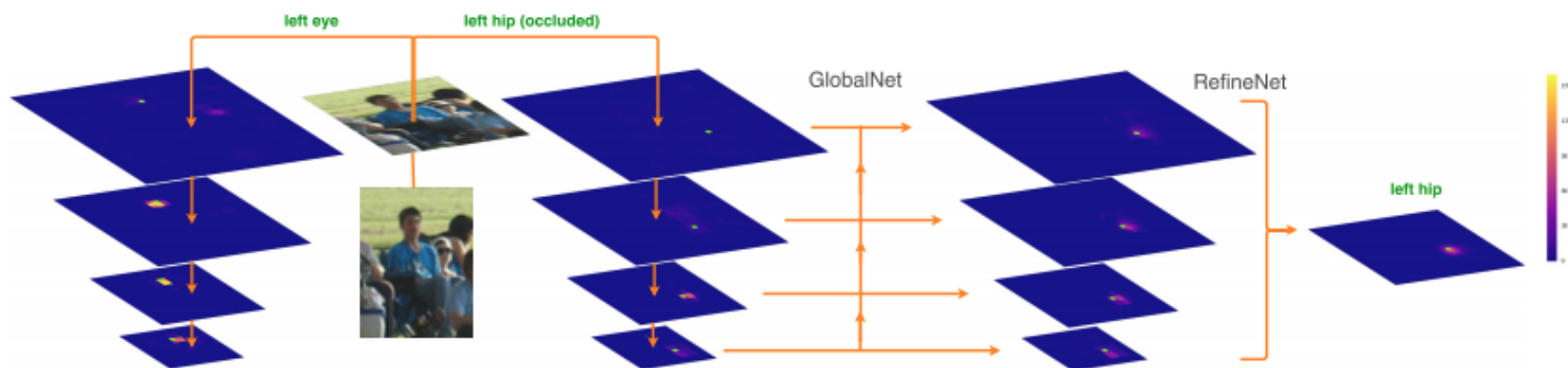


Figure 2. Output heatmaps from different features. The green dots means the groundtruth location of keypoints.

# Approach

- RefineNet
  - Concatenate all the pyramid features
  - Stack more bottleneck blocks into deeper layers, whose smaller spatial size achieves a good trade-off between effectiveness and efficiency
  - Online hard keypoints mining

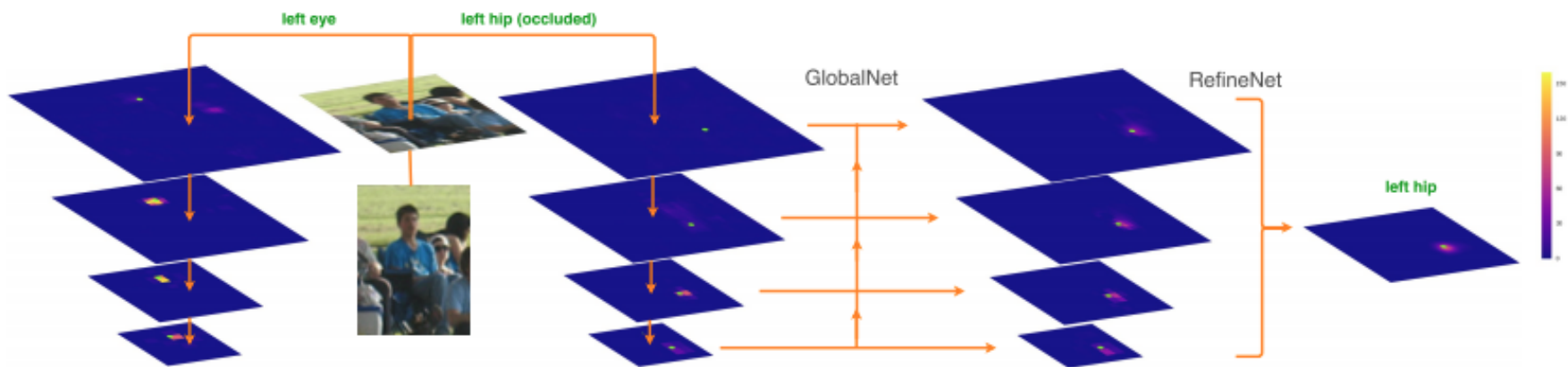


Figure 2. Output heatmaps from different features. The green dots means the groundtruth location of keypoints.



# Experiment

- Experimental Setup
  - Dataset and Evaluation Metric
    - Train: MS COCO trainval dataset  
(57K, 150K person instance)
    - Validation: MS COCO minival dataset(5000)
    - Test: test-dev(20K), test-challenge set(20K)
    - OKS(object keypoints similarity) based mAP
  - Cropping Strategy
  - Data Augmentation Strategy
  - Training Details
  - Testing Details

# Experiment

- Ablation Experiment
  - Person Detector
    - Non-Maximum Suppression (NMS) strategies
    - Detection Performance
  - Cascaded Pyramid Network
    - Design Choices of RefineNet
  - Online Hard Keypoints Mining
    - The loss function of GlobalNet: L2 loss
    - Only punish the top  $M$  ( $M < N$ ) keypoint losses out of  $N$

$M$	6	8	10	12	14	17
AP (OKS)	68.8	69.4	69.0	69.0	69.0	68.6

- Data Pre-processing
- Results on MS COCO Keypoints Challenge

# Conclusion

- Cascade Pyramid Network (CPN) is presented to address the 'hard' keypoints

```

9  class CPN(nn.Module):
10     def __init__(self, resnet, output_shape, num_class, pretrained=True):
11         super(CPN, self).__init__()
12         channel_settings = [2048, 1024, 512, 256]
13         self.resnet = resnet
14         self.global_net = globalNet(channel_settings, output_shape, num_class)
15         self.refine_net = refineNet(channel_settings[-1], output_shape, num_class)
16
17     def forward(self, x):
18         res_out = self.resnet(x)
19         global_fms, global_outs = self.global_net(res_out)
20         refine_out = self.refine_net(global_fms)
21
22         return global_outs, refine_out
23
24     def CPN50(out_size, num_class, pretrained=True):
25         res50 = resnet50(pretrained=pretrained)
26         model = CPN(res50, output_shape=out_size, num_class=num_class, pretrained=pretrained)
27         return model
28
29     def CPN101(out_size, num_class, pretrained=True):
30         res101 = resnet101(pretrained=pretrained)
31         model = CPN(res101, output_shape=out_size, num_class=num_class, pretrained=pretrained)
32         return model

```

---

```

96 class ResNet(nn.Module):
97
98     def __init__(self, block, layers, num_classes=1000):
99         self.inplanes = 64
100         super(ResNet, self).__init__()
101         self.conv1 = nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3,
102                                bias=False)
103         self.bn1 = nn.BatchNorm2d(64)
104         self.relu = nn.ReLU(inplace=True)
105         self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
106         self.layer1 = self._make_layer(block, 64, layers[0])
107         self.layer2 = self._make_layer(block, 128, layers[1], stride=2)
108         self.layer3 = self._make_layer(block, 256, layers[2], stride=2)
109         self.layer4 = self._make_layer(block, 512, layers[3], stride=2)
110
111         for m in self.modules():
112             if isinstance(m, nn.Conv2d):
113                 n = m.kernel_size[0] * m.kernel_size[1] * m.out_channels
114                 m.weight.data.normal_(0, math.sqrt(2. / n))
115             elif isinstance(m, nn.BatchNorm2d):
116                 m.weight.data.fill_(1)
117                 m.bias.data.zero_()
118
119     def _make_layer(self, block, planes, blocks, stride=1):
120         downsample = None
121         if stride != 1 or self.inplanes != planes * block.expansion:
122             downsample = nn.Sequential(
123                 nn.Conv2d(self.inplanes, planes * block.expansion,
124                           kernel_size=1, stride=stride, bias=False),
125                 nn.BatchNorm2d(planes * block.expansion),
126             )
127
128         layers = []
129         layers.append(block(self.inplanes, planes, stride, downsample))
130         self.inplanes = planes * block.expansion
131         for i in range(1, blocks):
132             layers.append(block(self.inplanes, planes))
133
134         return nn.Sequential(*layers)
135
136     def forward(self, x):
137         x = self.conv1(x)
138         x = self.bn1(x)
139         x = self.relu(x)
140         x = self.maxpool(x)
141
142         x1 = self.layer1(x)
143         x2 = self.layer2(x1)
144         x3 = self.layer3(x2)
145         x4 = self.layer4(x3)
146
147         return [x4, x3, x2, x1]

```

```

1  import torch.nn as nn
2  import torch
3  import math
4
5  class globalNet(nn.Module):
6      def __init__(self, channel_settings, output_shape, num_class):
7          super(globalNet, self).__init__()
8          self.channel_settings = channel_settings
9          laterals, upsamples, predict = [], [], []
10         for i in range(len(channel_settings)):
11             laterals.append(self._lateral(channel_settings[i]))
12             predict.append(self._predict(output_shape, num_class))
13             if i != len(channel_settings) - 1:
14                 upsamples.append(self._upsample())
15         self.laterals = nn.ModuleList(laterals)
16         self.upsamples = nn.ModuleList(upsamples)
17         self.predict = nn.ModuleList(predict)
18
19         for m in self.modules():
20             if isinstance(m, nn.Conv2d):
21                 n = m.kernel_size[0] * m.kernel_size[1] * m.out_cha
22                 m.weight.data.normal_(0, math.sqrt(2. / n))
23                 if m.bias is not None:
24                     m.bias.data.zero_()
25             elif isinstance(m, nn.BatchNorm2d):
26                 m.weight.data.fill_(1)
27                 m.bias.data.zero_()
28
29         def _lateral(self, input_size):
30             layers = []
31             layers.append(nn.Conv2d(input_size, 256,
32                                   kernel_size=1, stride=1, bias=False))
33             layers.append(nn.BatchNorm2d(256))
34             layers.append(nn.ReLU(inplace=True))
35
36             return nn.Sequential(*layers)

```

```

37
38         def _upsample(self):
39             layers = []
40             layers.append(torch.nn.Upsample(scale_factor=2, mode='bilinear', align_corners=True))
41             layers.append(torch.nn.Conv2d(256, 256,
42                                           kernel_size=1, stride=1, bias=False))
43             layers.append(nn.BatchNorm2d(256))
44
45             return nn.Sequential(*layers)
46
47         def _predict(self, output_shape, num_class):
48             layers = []
49             layers.append(nn.Conv2d(256, 256,
50                                   kernel_size=1, stride=1, bias=False))
51             layers.append(nn.BatchNorm2d(256))
52             layers.append(nn.ReLU(inplace=True))
53
54             layers.append(nn.Conv2d(256, num_class,
55                                   kernel_size=3, stride=1, padding=1, bias=False))
56             layers.append(nn.Upsample(size=output_shape, mode='bilinear', align_corners=True))
57             layers.append(nn.BatchNorm2d(num_class))
58
59             return nn.Sequential(*layers)
60
61         def forward(self, x):
62             global_fms, global_outs = [], []
63             for i in range(len(self.channel_settings)):
64                 if i == 0:
65                     feature = self.laterals[i](x[i])
66                 else:
67                     feature = self.laterals[i](x[i]) + up
68                 global_fms.append(feature)
69                 if i != len(self.channel_settings) - 1:
70                     up = self.upsamples[i](feature)
71                 feature = self.predict[i](feature)
72                 global_outs.append(feature)
73
74             return global_fms, global_outs

```

```

3
4 class Bottleneck(nn.Module):
5     expansion = 4
6
7     def __init__(self, inplanes, planes, stride=1):
8         super(Bottleneck, self).__init__()
9         self.conv1 = nn.Conv2d(inplanes, planes, kernel_size=1, bias=False)
10        self.bn1 = nn.BatchNorm2d(planes)
11        self.conv2 = nn.Conv2d(planes, planes, kernel_size=3, stride=stride,
12                                padding=1, bias=False)
13        self.bn2 = nn.BatchNorm2d(planes)
14        self.conv3 = nn.Conv2d(planes, planes * 2, kernel_size=1, bias=False)
15        self.bn3 = nn.BatchNorm2d(planes * 2)
16        self.relu = nn.ReLU(inplace=True)
17
18        self.downsample = nn.Sequential(
19            nn.Conv2d(inplanes, planes * 2,
20                    kernel_size=1, stride=stride, bias=False),
21            nn.BatchNorm2d(planes * 2),
22        )
23
24        self.stride = stride
25
26    def forward(self, x):
27        residual = x
28
29        out = self.conv1(x)
30        out = self.bn1(out)
31        out = self.relu(out)
32
33        out = self.conv2(out)
34        out = self.bn2(out)
35        out = self.relu(out)
36
37        out = self.conv3(out)
38        out = self.bn3(out)
39
40        if self.downsample is not None:
41            residual = self.downsample(x)
42
43        out += residual
44        out = self.relu(out)
45
46    return out
47
48 class refineNet(nn.Module):
49     def __init__(self, lateral_channel, out_shape, num_class):
50         super(refineNet, self).__init__()
51         cascade = []
52         num_cascade = 4
53         for i in range(num_cascade):
54             cascade.append(self._make_layer(lateral_channel, num_cascade-i-1, out_shape))
55         self.cascade = nn.ModuleList(cascade)
56         self.final_predict = self._predict(4*lateral_channel, num_class)
57
58     def _make_layer(self, input_channel, num, output_shape):
59         layers = []
60         for i in range(num):
61             layers.append(Bottleneck(input_channel, 128))
62             layers.append(nn.Upsample(size=output_shape, mode='bilinear', align_corners=True))
63         return nn.Sequential(*layers)
64
65     def _predict(self, input_channel, num_class):
66         layers = []
67         layers.append(Bottleneck(input_channel, 128))
68         layers.append(nn.Conv2d(256, num_class,
69                                 kernel_size=3, stride=1, padding=1, bias=False))
70         layers.append(nn.BatchNorm2d(num_class))
71         return nn.Sequential(*layers)
72
73     def forward(self, x):
74         refine_fms = []
75         for i in range(4):
76             refine_fms.append(self.cascade[i](x[i]))
77         out = torch.cat(refine_fms, dim=1)
78         out = self.final_predict(out)
79         return out

```