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 ROIPooling in FPN

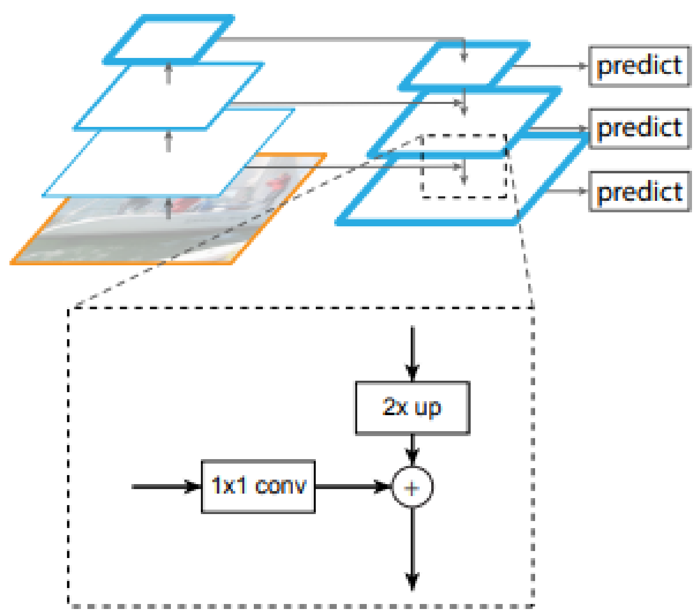

Figure 3. A building block illustrating the lateral connection and the top-down pathway, merged by addition.

Approach

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

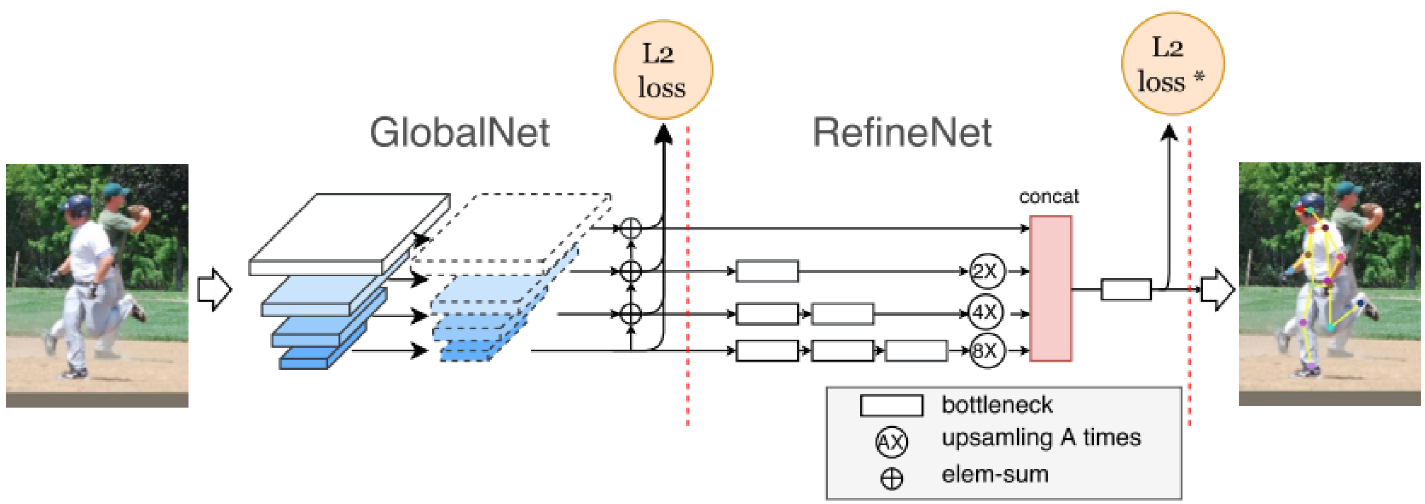

Figure 1. Cascaded Pyramid Network. “L2 loss*” means L2 loss with online hard keypoints mining.

Approach

– GlobalNet
  - Based one 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

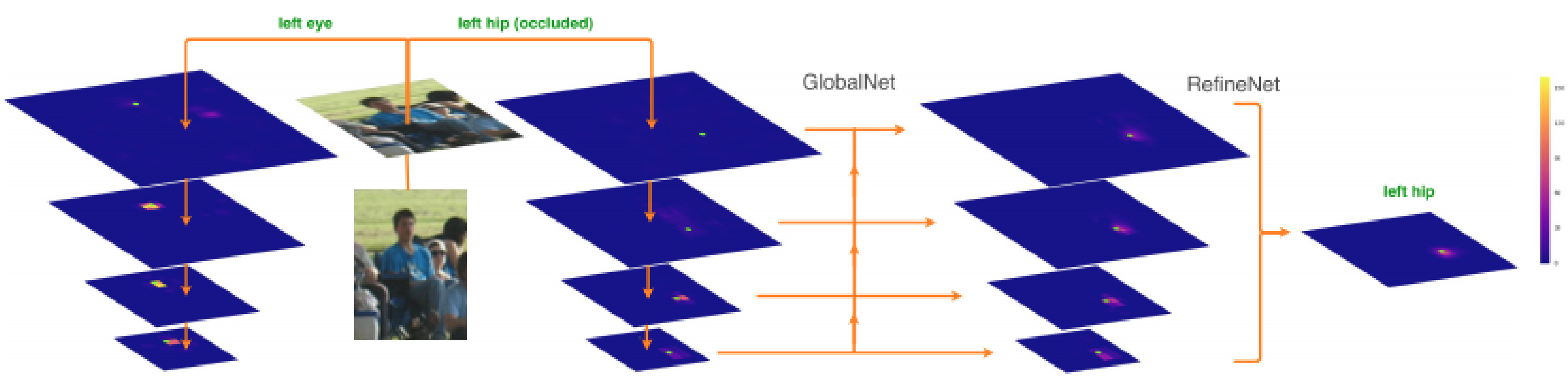

Figure 2. Output heatmaps from different features. The green dots means the groundtruth location of keypoints.

Approach

- GlobalNet
  - U-shape structure + FPN = feature pyramid structure
  - 1x1 convolutional kernel

    → element-wise sum (in upsampling)

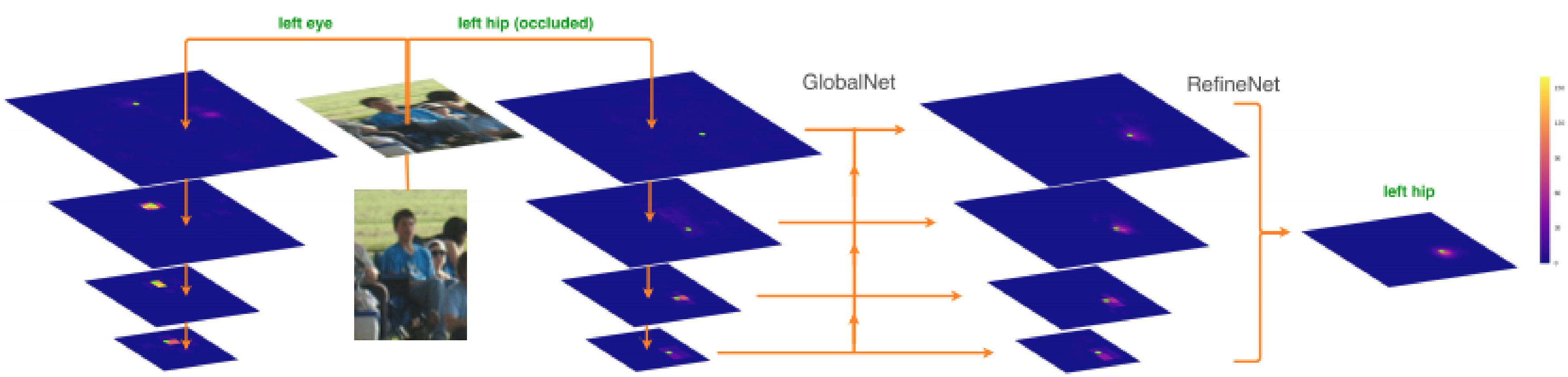

Figure 2. Output heatmaps from different features. The green dots means the groundtruth location of keypoints.

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**

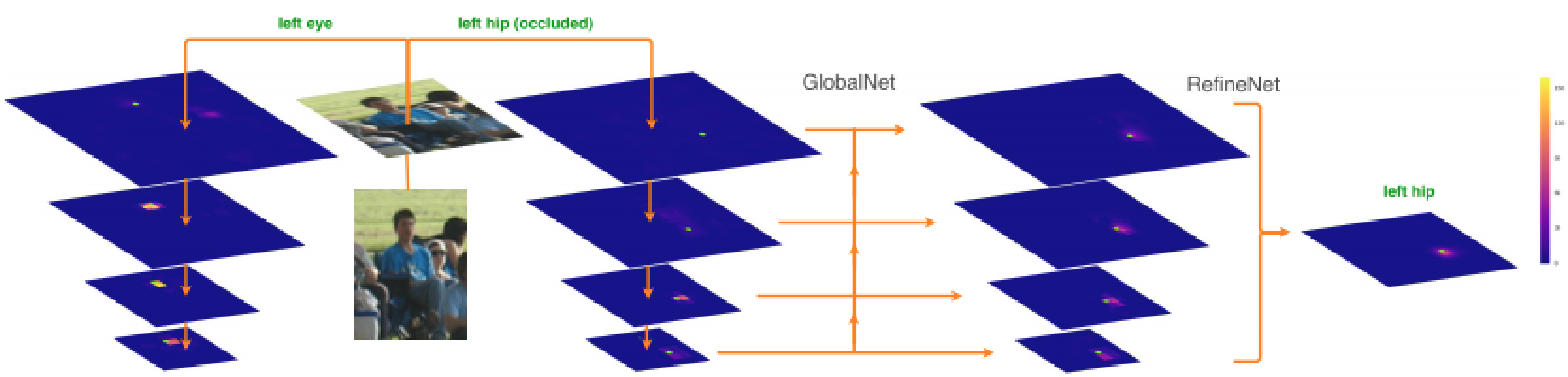

Figure 2. Output heatmaps from different features. The green dots means the groundtruth location of keypoints.

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

```
class CPN(nn.Module):
    def __init__(self, resnet, output_shape, num_class, pretrained=True):
        super(CPN, self).__init__()
        channel_settings = [2048, 1024, 512, 256]
        self.resnet = resnet
        self.global_net = globalNet(channel_settings, output_shape, num_class)
        self.refine_net = refineNet(channel_settings[-1], output_shape, num_class)

    def forward(self, x):
        res_out = self.resnet(x)
        global_fms, global_outs = self.global_net(res_out)
        refine_out = self.refine_net(global_fms)

        return global_outs, refine_out

def CPN50(out_size,num_class,pretrained=True):
    res50 = resnet50(pretrained=pretrained)
    model = CPN(res50, output_shape=out_size,num_class=num_class, pretrained=pretrained)
    return model

def CPN101(out_size,num_class,pretrained=True):
    res101 = resnet101(pretrained=pretrained)
    model = CPN(res101, output_shape=out_size,num_class=num_class, pretrained=pretrained)
    return model
```

https://github.com/GengDavid/pytorch-cpn/blob/master/networks

```python
class ResNet(nn.Module):

    def __init__(self, block, layers, num_classes=1000):
        self.inplanes = 64
        super(ResNet, self).__init__()
        self.conv1 = nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3,
                               bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.layer1 = self._make_layer(block, 64, layers[0])
        self.layer2 = self._make_layer(block, 128, layers[1], stride=2)
        self.layer3 = self._make_layer(block, 256, layers[2], stride=2)
        self.layer4 = self._make_layer(block, 512, layers[3], stride=2)

        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                n = m.kernel_size[0] * m.kernel_size[1] * m.out_channels
                m.weight.data.normal_(0, math.sqrt(2. / n))
            elif isinstance(m, nn.BatchNorm2d):
                m.weight.data.fill_(1)
                m.bias.data.zero_()
```

```python
    def _make_layer(self, block, planes, blocks, stride=1):
        downsample = None
        if stride != 1 or self.inplanes != planes * block.expansion:
            downsample = nn.Sequential(
                nn.Conv2d(self.inplanes, planes * block.expansion,
                          kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(planes * block.expansion),
            )

        layers = []
        layers.append(block(self.inplanes, planes, stride, downsample))
        self.inplanes = planes * block.expansion
        for i in range(1, blocks):
            layers.append(block(self.inplanes, planes))

        return nn.Sequential(*layers)

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)

        x1 = self.layer1(x)
        x2 = self.layer2(x1)
        x3 = self.layer3(x2)
        x4 = self.layer4(x3)

        return [x4, x3, x2, x1]
```

https://github.com/GengDavid/pytorch-cpn/blob/master/networks

```python
import torch.nn as nn
import torch
import math

class globalNet(nn.Module):
    def __init__(self, channel_settings, output_shape, num_class):
        super(globalNet, self).__init__()
        self.channel_settings = channel_settings
        laterals, upsamples, predict = [], [], []
        for i in range(len(channel_settings)):
            laterals.append(self._lateral(channel_settings[i]))
            predict.append(self._predict(output_shape, num_class))
            if i != len(channel_settings) - 1:
                upsamples.append(self._upsample())
        self.laterals = nn.ModuleList(laterals)
        self.upsamples = nn.ModuleList(upsamples)
        self.predict = nn.ModuleList(predict)

        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                n = m.kernel_size[0] * m.kernel_size[1] * m.out_channels
                m.weight.data.normal_(0, math.sqrt(2. / n))
                if m.bias is not None:
                    m.bias.data.zero_()
            elif isinstance(m, nn.BatchNorm2d):
                m.weight.data.fill_(1)
                m.bias.data.zero_()

    def _lateral(self, input_size):
        layers = []
        layers.append(nn.Conv2d(input_size, 256,
            kernel_size=1, stride=1, bias=False))
        layers.append(nn.BatchNorm2d(256))
        layers.append(nn.ReLU(inplace=True))

        return nn.Sequential(*layers)
```

```python
    def _upsample(self):
        layers = []
        layers.append(torch.nn.Upsample(scale_factor=2, mode='bilinear', align_corners=True))
        layers.append(torch.nn.Conv2d(256, 256,
            kernel_size=1, stride=1, bias=False))
        layers.append(nn.BatchNorm2d(256))

        return nn.Sequential(*layers)

    def _predict(self, output_shape, num_class):
        layers = []
        layers.append(nn.Conv2d(256, 256,
            kernel_size=1, stride=1, bias=False))
        layers.append(nn.BatchNorm2d(256))
        layers.append(nn.ReLU(inplace=True))

        layers.append(nn.Conv2d(256, num_class,
            kernel_size=3, stride=1, padding=1, bias=False))
        layers.append(nn.Upsample(size=output_shape, mode='bilinear', align_corners=True))
        layers.append(nn.BatchNorm2d(num_class))

        return nn.Sequential(*layers)

    def forward(self, x):
        global_fms, global_outs = [], []
        for i in range(len(self.channel_settings)):
            if i == 0:
                feature = self.laterals[i](x[i])
            else:
                feature = self.laterals[i](x[i]) + up
            global_fms.append(feature)
            if i != len(self.channel_settings) - 1:
                up = self.upsamples[i](feature)
            feature = self.predict[i](feature)
            global_outs.append(feature)

        return global_fms, global_outs
```

https://github.com/GengDavid/pytorch-cpn/blob/master/networks

```python
class Bottleneck(nn.Module):
    expansion = 4

    def __init__(self, inplanes, planes, stride=1):
        super(Bottleneck, self).__init__()
        self.conv1 = nn.Conv2d(inplanes, planes, kernel_size=1, bias=False)
        self.bn1 = nn.BatchNorm2d(planes)
        self.conv2 = nn.Conv2d(planes, planes, kernel_size=3, stride=stride,
                               padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(planes)
        self.conv3 = nn.Conv2d(planes, planes * 2, kernel_size=1, bias=False)
        self.bn3 = nn.BatchNorm2d(planes * 2)
        self.relu = nn.ReLU(inplace=True)

        self.downsample = nn.Sequential(
                nn.Conv2d(inplanes, planes * 2,
                          kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(planes * 2),
            )

        self.stride = stride

    def forward(self, x):
        residual = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)
        out = self.relu(out)

        out = self.conv3(out)
        out = self.bn3(out)

        if self.downsample is not None:
            residual = self.downsample(x)

        out += residual
        out = self.relu(out)

        return out
```

```python
class refineNet(nn.Module):
    def __init__(self, lateral_channel, out_shape, num_class):
        super(refineNet, self).__init__()
        cascade = []
        num_cascade = 4
        for i in range(num_cascade):
            cascade.append(self._make_layer(lateral_channel, num_cascade-i-1, out_shape))
        self.cascade = nn.ModuleList(cascade)
        self.final_predict = self._predict(4*lateral_channel, num_class)

    def _make_layer(self, input_channel, num, output_shape):
        layers = []
        for i in range(num):
            layers.append(Bottleneck(input_channel, 128))
        layers.append(nn.Upsample(size=output_shape, mode='bilinear', align_corners=True))
        return nn.Sequential(*layers)

    def _predict(self, input_channel, num_class):
        layers = []
        layers.append(Bottleneck(input_channel, 128))
        layers.append(nn.Conv2d(256, num_class,
            kernel_size=3, stride=1, padding=1, bias=False))
        layers.append(nn.BatchNorm2d(num_class))
        return nn.Sequential(*layers)

    def forward(self, x):
        refine_fms = []
        for i in range(4):
            refine_fms.append(self.cascade[i](x[i]))
        out = torch.cat(refine_fms, dim=1)
        out = self.final_predict(out)
        return out
```

https://github.com/GengDavid/pytorch-cpn/blob/master/networks