Compensated Convolutional Neural Network for Object Localization and Semantic Segmentation

Xiaoqin Wang

Abstract

The class activation map (CAM) generated by the convolutional neural classification network has been widely used for weakly supervised object localization (WSOL) and semantic segmentation (WSSS) tasks. However, such convolutional neural classification network usually focus on discriminative object regions. To mitigate this problem, in this report, we propose a Compensated Convolutional Neural Network (CCNN) for the WSOL and WSSS tasks. We utilize an attention modulation module (AMM) to rearrange the distribution of feature importance from the channel-spatial sequential perspective. AMM helps to explicitly model channel-wise interdependencies and spatial encodings to adaptively modulate localization- and segmentation-oriented activation responses. Experimental results on CUB-200-2011 and PASCAL VOC2012 show that our improved convolutional neural network approach can obtain better performance than the traditional convolutional neural network.

Keywords: object localization, semantic segmentation, activation modulation.

1 Introduction

Object localization and semantic segmentation are the fundamental and crucial tasks due to extensive applications in the field of computer vision. Massive image data and granular pixel-level annotations are usually required for object localization and semantic segmentation. However, it is time-consuming and labor-intensive to obtain bounding box or pixel-level annotations. To alleviate such expensive and unwieldy annotations, many works tend to resort to weakly supervised manner, such as bounding boxes supervision [1], scribbles supervision [2], point supervision [3], and image-level supervision [4].

Image-level supervision is an exceedingly favorable scheme since such coarse are more readily available in practice. Most previous weakly supervised object localization and semantic segmentation methods rely on the class activation map (CAM) [5], which is model based on Convolutional Neural Network (CNN), to estimate the location of the target object. With image-level supervision, the classifier tries to find discriminative regions of target objects. Thus, though image-level labels enable CAM to indicate the correct location of target objects, they also limit the focus of CAM on sparse and discriminative object regions. As a result, it could be very difficult to precisely estimate complete object regions by directly applying CAM to WSOL and WSSS.

In this report, we try to improve the traditional convolutional neural network based on task-specific concepts, and then we propose a Compensated Convolutional Neural Network (CCNN) as the feature extraction network for weakly supervised object localization and semantic segmentation tasks. Specifically.

we exploit an attention modulation module (AMM) [6] to rearrange the distribution of feature importance from the channel-spatial sequential perspective. AMM can help to explicitly model channel-wise interdependencies and spatial encodings to adaptively modulate localization- and segmentation-oriented activation repspons.

2 Related Work

Most of the existing object localization and semantic segmentation works consider the image-level labels to extract the class activation map based on classification convolutional neural network. The most representative work is the Contrastive learning of Class-agnostic Activation Map (CCAM) [7], which is also the work that we want to re-implementation. In this section, we will mainly introduce the backbone network, the framework and the loss functions of the CCAM model.

2.1 The Framework of CCAM

The overall network architecture of CCAM is shown in Figure 1. Given a batch of n images $X_{1:n} = \{X_i\}_{i=1}^n$, the encoder $h(\cdot)$ maps X to high-level feature maps $Z_{1:n} = \{Z_i\}_{i=1}^n$, in which $Z_i \in R^{C\times H\times W}$. C and $H \times W$ denote the channel number and spatial dimension, respectively. The popular convolutional neural network ResNet is used as the encoder $h(\cdot)$. Supervised or unsupervised pretraining, e.g., moco, on ImageNet-1K can be adopted as initialization of $h(\cdot)$. Based on the extracted feature maps Z, the disentangler employs an activation head $\varphi(\cdot)$ to produce the class-agnostic activation maps $P_{1:n} = \{P_i\}_{i=1}^n$, in which $P_i \in R^{C\times H\times W}$. Specifically, $\varphi(\cdot)$ is a 3×3 convolution with a batch normalization layer. Suppose P_i activates foreground regions, the background activation map of the i-th sample can be formulated as $(1 - P_i)$. The foreground and background activation maps can finally disentangle the feature map $Z_{1:n}$ into the foreground and background feature representations, i.e., V_i^f and , respectively. For the i-th sample, V_i^f and V_i^b can be derived as:

$$V_i^f = P_i \otimes Z_i^T, V_i^b = (1 - P_i) \otimes Z_i^T. \tag{1}$$

Here, P_i and Z_i are flattened, i.e., $P_i \in R^{1\times HW}$ and $Z_i \in R^{1\times HW}$. $V_i^f \in R^{1\times C}$ and $V_i^f \in R^{1\times C}$. $\otimes$ and T indicate the matrix multiplication and transpose, respectively.

2.2 The loss functions of CCAM

The CCAM applies the contrastive loss to pull close and push apart the representations in positive and negative pairs, and then the class-agnostic activation map gradually separates the regions of foreground object and background in the image. The contrastive loss L of CCAM, which is formulated as the summation of L_{pos} and L_{neg}:

$$L = L_{neg} + L_{pos}, \tag{2}$$

where L_{neg} is the cross-image foreground-background contrast, which locates the foreground object regions by only utilizing the semantic information among foreground and background representations.

The negative contrastive loss is designed as:

$$L_{neg} = -\frac{1}{n^2}\sum_{i=1}^{n}\sum_{j=1}^{n} log(1 - S_{i,j}^{neg}), \tag{3}$$

where $S_{i,j}^{neg} = sim(V_i^f, V_j^b)$ is the consine similarity between V_i^f and V_j^b. L_{neg} considers the contrastive comparisons both within image ($i = j$) and cross image ($i \neq j$). In addition, the positive contrastive loss is formulated as:

$$L_{pos} = L_{pos}^f + L_{pos}^b, \tag{4}$$

where L_{pos}^f and L_{pos}^b can be designed as:

$$L_{pos}^f = -\frac{1}{n(n-1)}\sum_{i=1}^{n} I_{[i\neq j]}(w_{i,j}^f \cdot log(S_{i,j}^f)), \tag{5}$$

$$L_{pos}^b = -\frac{1}{n(n-1)}\sum_{i=1}^{n} I_{[i\neq j]}(w_{i,j}^b \cdot log(S_{i,j}^b)), \tag{6}$$

where $I_{[i\neq j]} \in 0, 1$ is an indicator function and it equals 1 if $i \neq j$.

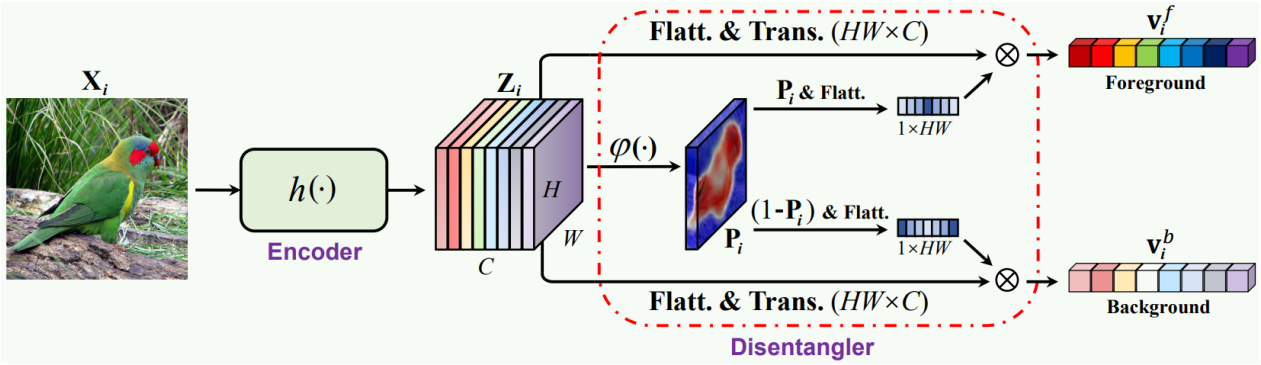

Figure 1: The overall network architecture of the contrastive learning of class-agnostic activation map. The encoder network $h(\cdot)$ maps image X_i to the feature map Z_i. In disentangler, the activation head $\varphi(\cdot)$ produces the class-agnostic activation map P_i. Suppose P_i activates the foreground regions and the backgournd activation map can be dervied as $(1 - P_i)$. Based on the foreground and background activation maps, Z_i can be disentangled into the foreground and background feature representations, i.e., V_i^f and V_i^b. In evaluation, only the trained $h(\cdot)$ and $\varphi(\cdot)$ are used for generating the class-agnostic activation map P_i. Flatt. is the matrix flattening, Trans. is the matrix transpose, and the $\otimes$ is the matrix multiplication.

3 Methodology

In this section, we first briefly introduce the attention modulation module (AMM). Then we illustrate and improved method of CCAM, which is named Compensated Convolutional Neural Network (CCNN). Finally, how to use the CCNN for WSOL and WSSS are illustrated.

3.1 AMM module

The attention modulation module (AMM) is used to assist the compensation module to extract more regions essential for object localization and semantic segmentation tasks. As shown in Figure 2, AMM consists of channel attention modulation and spatial attention modulation.

For the channel attention, We firstly feed features $F(I)$ to the channel attention. The channel interdependencies are explicitly modeled by the average pooling and the convolutional layer, which

reflect the sensitivity to informative features. Then, we exploit the Gaussian function to enhance the minor features and restrain the most and least sensitive features. Finally, we conduct an element-wise multiplication between the channel attention maps and the input feature maps to generate the redistributed features, which is defined as:

$$F_c(I) = \psi(G(H(P_s(F(I))))) \otimes F(I), \tag{7}$$

where $\psi(\cdot)$ denotes the channel attention maps which are expanded to the dimensions of feature maps, $G(\cdot)$ is the Gaussian function, which is leveraged to reassign the distribution of features to highlight the minor features in the channel dimension, $H(\cdot)$ is the convolution layer, and $P_s(\cdot)$ is the spatial average pooling function.

To further model inter-spatial relationships in the spatial dimension, we also introduce spatial attention to cascade after channel attention. Specifically, we first employ a channel average pooling P_c on F_c in the channel dimension and then apply a convolution operation H to them. The output feature maps illustrate the importance of the features in the spatial dimensions. Then we perform the Gaussian function on the output feature maps to increase the minor activation. Finally, we make an element-wise multiplication between the spatial attention maps and the feature maps. The implementation process can formulate as:

$$F_s(I) = \psi(G(H(P_c(F_c(I))))) \otimes F_c(I), \tag{8}$$

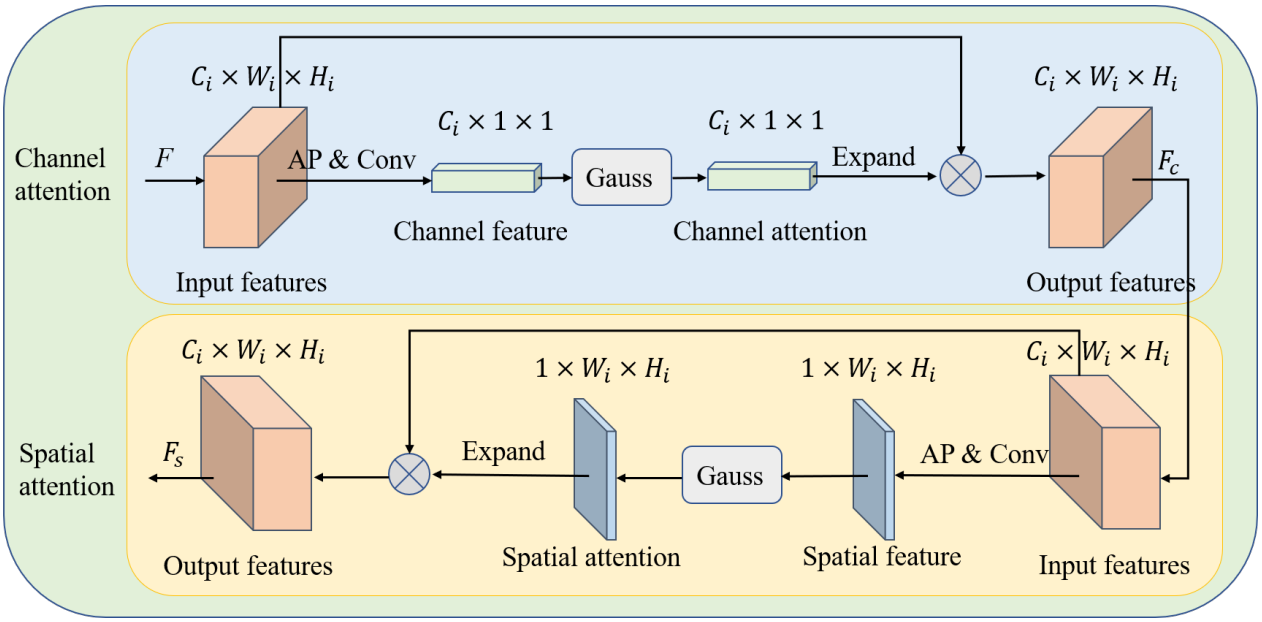

Figure 2: Illustration of the attention modulation module, which aims to modulate the activation maps of features in the channel-spatial sequential manner. It consists of channel attention and spatial attention. *AP* represents the average pooling layer, *Conv* is the convolutional layer, *Gauss* is the gaussian function, and $\otimes$ is the element-wise multiplication.

3.2 CCNN module

Most existing methods resort to classification-based Class Activation Maps (CAMs) to play as the initial pseudo labels, which tend to focus on the discriminative image regions and lack customized characteristics for object localization and semantic segmentation. To mitigate this problem, we incorporate the AMM module as the compensated module into the ResNet50 network structure, and propose a

novel convolutional neural network, named Compensated Convolutional Neural Network (CCNN), as shown in Figure 3. Specifically, We integrate the AMM module into stage 4 of ResNet as the residual AMM block, which can be treated as the compensated module. Then, we retain the stages 0, 1, 2, 3, and 4 of the ResNet50, and then link the residual AMM after stage 3. Finally, we can obtain feature maps Z_i of the input X_i by concatenating all channel feature maps from stages 3, 4 and the residual AMM, which can be defined as:

$$Z_i = F_{ccnn}(X_i) = \Omega(S_3(F_2), S_4(F_3), R_{amm}(F_3)), \tag{9}$$

where Ω is the Concatenate operation, $S_i(\cdot), i \in (0, 1, 2, 3, 4)$ is the function of the corresponding stage, the $F_i, i \in (0, 1, 2, 3, 4)$ is the output of the corresponding stage, $R_{amm}(\cdot)$ is the function of residual AMM module. The final output feature maps of the CCNN module is $F_{ccnn}(X_i)$.

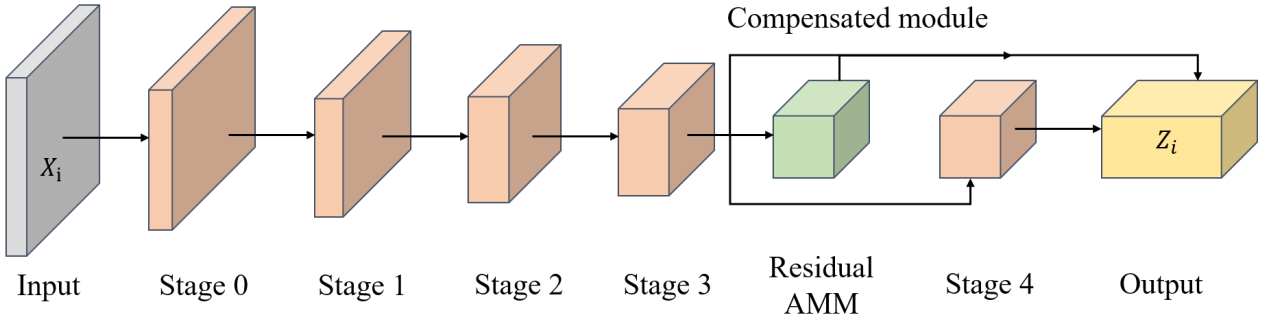

Figure 3: Illustration of the CCNN module. The backbone is the ResNet50 model. Stages of 0, 1, 2, 3 and 4 come from the initial ResNet50. The Residual AMM is formed by the stage 4 from ResNet50 and the AMM module. The output feature maps are Cascaded by the output of stage 3, stage 4 and the residual AMM.

3.3 The framework of CCNN

The backbone of the CCAM method only identifies the most discriminative region of objects. This will fundamentally lose the performance of the extracted CAM. Therefore, we replace the backbone of the ResNet50 by our CCNN module to construct a novel framework of CCNN for object localization and semantic segmentation tasks, which is shown in Figure 4. Specifically, the image X_i is input into the CCNN network, and then we obtain the feature maps Z_i. We follow the CCAM to disentangle the foreground object and background cues. It is worth mentioning that our overall framework uses the same loss functions as in CCAM.

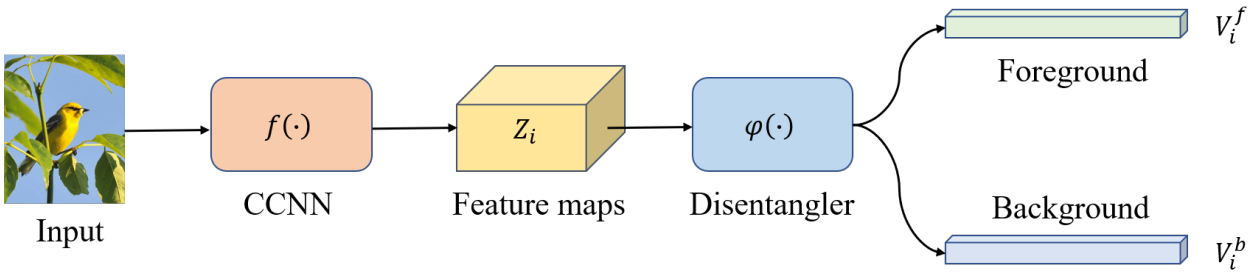

Figure 4: The overall network architecture of CCNN. $f(\cdot)$ is the function of CCNN module, and the $\varphi(\cdot)$ is the disentangler module.

The CCNN network alleviates the task gap issue of using classification-based CAMs to perform localization and segmentation tasks in previous work, which contributes to providing more task-special

cues. It can dig out the essential regions that are easily ignored by the ResNet50 network.

3.4 Application of CCNN

We use our proposed CCNN framework for the weakly supervised object localiation and semantic segmentation tasks.

3.4.1 CCNN for WSOL

We follow the route of PSOL [8] to adapt our CCNN to WSOL. Specifically, WSOL is divided into two tasks: class-agnostic object localization and object classification. The proposed method directly learns class-agnostic activation maps from the whole dataset based on the CCNN network. We set a threshold to binarize the class-agnostic activation maps and then extract class-agnostic object bounding boxes as pseudo labels. The localization model is trained with these pseudo labels for object bounding box predictions. The popular network, e.g., EfficientNet [9], is adopted for object classification.

3.4.2 CCNN for WSSS

We first use CAM-based methods to generate the initial CAM for each image and then apply CCNN to refine it. Specifically, we use the background activation maps $(1 - P)$ as pseudo labels to further train a model to predict the background regions, i.e., background cues, in the image. We concatenate the predicted background cues with the initial CAM and perform the argmax process along the channel dimension to refine the initial CAM. We just use a simple way to demonstrate the effectiveness of our method for refinement of initial CAM.

4 The code of CCNN

In this section, we will show the key codes of the proposed CCNN method, including the AMM module and the CCNN network model.

4.1 AMM module

```
def GaussProjection(x, mean, std):
    sigma = math.sqrt(2 * math.pi) * std
    x_out = torch.exp(-(x - mean) ** 2 / (2 * std ** 2)) / sigma
    return x_out

class BasicConv(nn.Module):
    def __init__(self, in_planes, out_planes, kernel_size, stride=1, padding=0, dilation=1, groups
        =1, relu=True, bn=True, bias=False):
        super(BasicConv, self).__init__()
        self.out_channels = out_planes
        self.conv = nn.Conv2d(in_planes, out_planes, kernel_size=kernel_size, stride=stride, padding
            =padding, dilation=dilation, groups=groups, bias=bias)
        self.bn = nn.BatchNorm2d(out_planes,eps=1e-5, momentum=0.01, affine=True) if bn else None
        self.relu = nn.ReLU() if relu else None

    def forward(self, x):
        x = self.conv(x)
        if self.bn is not None:
            x = self.bn(x)
        if self.relu is not None:
            x = self.relu(x)
        return x
```

```
21
22 class Flatten(nn.Module):
23     def forward(self, x):
24         return x.view(x.size(0), -1)
25
26 class ChannelGate(nn.Module):
27     def __init__(self, gate_channels, reduction_ratio=16, pool_types=['avg']):
28         super(ChannelGate, self).__init__()
29         self.gate_channels = gate_channels
30         self.mlp = nn.Sequential(
31             Flatten(),
32             nn.Linear(gate_channels, gate_channels // reduction_ratio),
33             nn.ReLU(),
34             nn.Linear(gate_channels // reduction_ratio, gate_channels)
35             )
36         self.pool_types = pool_types
37
38     def forward(self, x):
39         channel_att_sum = None
40         for pool_type in self.pool_types:
41             if pool_type == 'avg':
42                 avg_pool = F.avg_pool2d(x, (x.size(2), x.size(3)), stride=(x.size(2), x.size(3)))
43                 channel_att_raw = self.mlp(avg_pool)
44             elif pool_type == 'max':
45                 max_pool = F.max_pool2d(x, (x.size(2), x.size(3)), stride=(x.size(2), x.size(3)))
46                 channel_att_raw = self.mlp(max_pool)
47
48             if channel_att_sum is None:
49                 channel_att_sum = channel_att_raw
50             else:
51                 channel_att_sum = channel_att_sum + channel_att_raw
52
53         # Gauss modulation
54         mean = torch.mean(channel_att_sum).detach()
55         std = torch.std(channel_att_sum).detach()
56         scale = GaussProjection(channel_att_sum, mean, std).unsqueeze(2).unsqueeze(3).expand_as(x)
57
58         return x * scale
59
60 class ChannelPool(nn.Module):
61
62     def forward(self, x):
63         return torch.cat((torch.max(x, 1)[0].unsqueeze(1), torch.mean(x, 1).unsqueeze(1)), dim=1)
64
65 class SpatialGate(nn.Module):
66     def __init__(self):
67         super(SpatialGate, self).__init__()
68         kernel_size = 7
69         self.pool = ChannelPool()
70         self.spatial = BasicConv(2, 1, kernel_size, stride=1, padding=(kernel_size-1) // 2, relu=
               False)
71
72     def forward(self, x):
73         x_pool = self.pool(x)
74         x_out = self.spatial(x_pool)
75
76         # Gauss modulation
77         mean = torch.mean(x_out).detach()
78         std = torch.std(x_out).detach()
79         scale = GaussProjection(x_out, mean, std)
80
81         return x * scale
82
83 class AMM(nn.Module):
84     def __init__(self, gate_channels, reduction_ratio=16, pool_types=['avg']):
85         super(AMM, self).__init__()
86         self.ChannelAMM = ChannelGate(gate_channels, reduction_ratio, pool_types)
87         self.SpatialAMM = SpatialGate()
88
```

```
    def forward(self, x):
        x_out = self.ChannelAMM(x)
        x_out = self.SpatialAMM(x_out)
        return x_out
```

4.2 CCNN module

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

    def __init__(self, inplanes, planes, stride=1, downsample=None, use_amm=False):
        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 * self.expansion, kernel_size=1, bias=False)
        self.bn3 = nn.BatchNorm2d(planes * self.expansion)
        self.relu = nn.ReLU(inplace=True)
        self.downsample = downsample
        self.stride = stride

        self.use_amm = use_amm
        if self.use_amm == True:
            self.amm = AMM(planes * 4, 16)

    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)

        if self.use_amm:
            out = self.amm(out)

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

        return out
#=============================================================================
class ResNet(nn.Module):
    def __init__(self, block, layers, stride=None, use_amm=False):
        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], stride=stride[0], use_amm=use_amm)
        self.layer2 = self._make_layer(block, 128, layers[1], stride=stride[1], use_amm=use_amm)
        self.layer3 = self._make_layer(block, 256, layers[2], stride=stride[2], use_amm=use_amm)
        self.layer4 = self._make_layer(block, 512, layers[3], stride=stride[3], use_amm=use_amm)
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512 * block.expansion, 1000)

        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_()

    def _make_layer(self, block, planes, blocks, stride=1, use_amm=False):
        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, use_amm=use_amm))
        self.inplanes = planes * block.expansion
        for i in range(1, blocks):
            layers.append(block(self.inplanes, planes, use_amm=use_amm))

        return nn.Sequential(*layers)

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

        x = self.layer1(x)
        x = self.amm1(x)
        x = self.layer2(x)
        x = self.amm2(x)
        x = self.layer3(x)
        x = self.amm3(x)
        x = self.layer4(x)
        x = self.amm4(x)

        x = self.avgpool(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)

        return x
#=========================================================================
class ResNetSeries(nn.Module):
    def __init__(self, pretrained):
        super(ResNetSeries, self).__init__()

        if pretrained == 'supervised':
            print(f'Loading supervised pretrained parameters!')
            self.resnet50 = resnet50(pretrained=True, use_amm=False)
            self.resnet50_2 = resnet50(pretrained=True, use_amm=True)

        elif pretrained == 'mocov2':
            print(f'Loading unsupervised {pretrained} pretrained parameters!')
            self.resnet50 = resnet50(pretrained=True, use_amm=False)
            checkpoint = torch.load('moco_r50_v2-e3b0c442.pth', map_location="cpu")
            self.resnet50.load_state_dict(checkpoint['state_dict'], strict=False)

            self.resnet50_2 = resnet50(pretrained=True, use_amm=True)
            checkpoint = torch.load('moco_r50_v2-e3b0c442.pth', map_location="cpu")
            self.resnet50_2.load_state_dict(checkpoint['state_dict'], strict=False)

        elif pretrained == 'detco':
            print(f'Loading unsupervised {pretrained} pretrained parameters!')
            self.resnet50 = resnet50(pretrained=True, use_amm=False)
            checkpoint = torch.load('detco_200ep.pth', map_location="cpu")
            self.resnet50.load_state_dict(checkpoint['state_dict'], strict=False)
```

```
            self.resnet50_2 = resnet50(pretrained=True, use_amm=True)
            checkpoint = torch.load('detco_200ep.pth', map_location="cpu")
            self.resnet50_2.load_state_dict(checkpoint['state_dict'], strict=False)
        else:
            raise NotImplementedError

        self.conv1 = self.resnet50.conv1
        self.bn1 = self.resnet50.bn1
        self.relu = self.resnet50.relu
        self.maxpool = self.resnet50.maxpool

        # ========================================
        self.stage1 = nn.Sequential(self.resnet50.layer1)
        self.stage2 = nn.Sequential(self.resnet50.layer2)
        self.stage3 = nn.Sequential(self.resnet50.layer3)
        self.stage4 = nn.Sequential(self.resnet50.layer4)

        # ========================================
        self.stage2_1 = nn.Sequential(self.resnet50_2.layer1)
        self.stage2_2 = nn.Sequential(self.resnet50_2.layer2)
        self.stage2_3 = nn.Sequential(self.resnet50_2.layer3)
        self.stage2_4 = nn.Sequential(self.resnet50_2.layer4)

    def forward(self, x):

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

        x = self.stage1(x)
        x = self.stage2(x)
        x = self.stage3(x)

        tmp = x

        x1 = self.stage4(x)

        x2 = self.stage2_4(tmp)

        return torch.cat([x2, x1, x], dim=1)
```

5 Experiments

5.1 Experimental Setup

We use the CUB-200-2011 [10] dataset for the weakly supervised object localization. CUB-200-2011 is a fine-grained classification dataset, which consists of 200 species of birds with 5,994 images for training and 5,794 images for test. In addition, we use the PASCAL VOC2012 [11] for weakly supervised semantic segmentation. PASCAL VOC2012 is a popular semantic segmentation dataset with 20 object categories. It consists of 1,464 image for training, 1,449 images for validation, and 1,456 images for test.

For the WSOL task, we use Top-1, Toc-5 and GT-known localization accuracy for evaluation. The MaxBoxAccV2 [12] is also adopted for averaging the performance across different bounding boxes ratios, i.e., 30%, 50%, and 70%. For the WSSS task, the Intersection over Union (IoU) and mean Intersection over Union (mIoU) is adopted as the evaluation metrics.

5.2 Results of WSOL

Table 1 compares the quality of class-agnostic bounding boxes generated by ResNet50 and the proposed CCNN. When considering unsupervised pretraining moco model, the GT-known localization accuracy of CCNN has high performance, which has improved 2.41% than ResNet50. When supervised pretraining on ImageNet-1K is adopted as initialization, class-agnostic bounding boxes generated by CCNN have a higher GT-known localization accuracy than ResNet50. This indicates that our method has a good capability to generate higher-quality class-agnostic bounding boxes than ResNet50.

Table 1: The GT-known localization accuracy of CCAM and the proposed CCNN on CUB-200-2011.

Methods	Initialization	GT-known
CCAM	moco	88.18
CCNN	moco	90.59
CCAM	supervised	90.11
CCNN	supervised	91.22

Table 2 compares the performance of CCAM and our proposed CCNN method on CUB-200-2011. Our method achieves the localization accuracy of Top-1 of 83.19%, Top-5 of 92.78%, and GT-known of 94.59%. We also provide the results of MaxBoxAccV2 metric, which obtains the performance of 85.68%. One can observe that our CCNN achieves significantly better performance than the initial model. It is worth mentioning that the CCAM experimental results are our re-implementation results.

Table 2: Comparison the performance between the CCAM and the CCNN on CUB-200-2011 test set.

Method	Loc Backbone	Class Backbone	Init.	Top-1	Top-5	GT-known	MaxBoxAccV2
CCAM	DenseNet161	EfficientNet-B7	moco	81.89	91.17	92.99	80.64
CCNN	DenseNet161	EfficientNet-B7	moco	83.19	92.78	94.59	85.68
CCAM	DenseNet161	EfficientNet-B7	supervised	82.65	92.16	94.03	83.91
CCNN	DenseNet161	EfficientNet-B7	supervised	83.70	93.30	95.18	85.76

We present a visual comparison of CAM between the proposed CCNN and the CCAM in Figure 5 to validate that more complete and correct foreground regions can be predicted in the class-agnostic activation maps generated by CCNN. Then, we also apply the CAMs generated by CCNN to WSOL. The visual comparison of the bounding boxes in Figure 6 shows that the CCNN can help to extract more accurate bounding boxes for the object localization task.

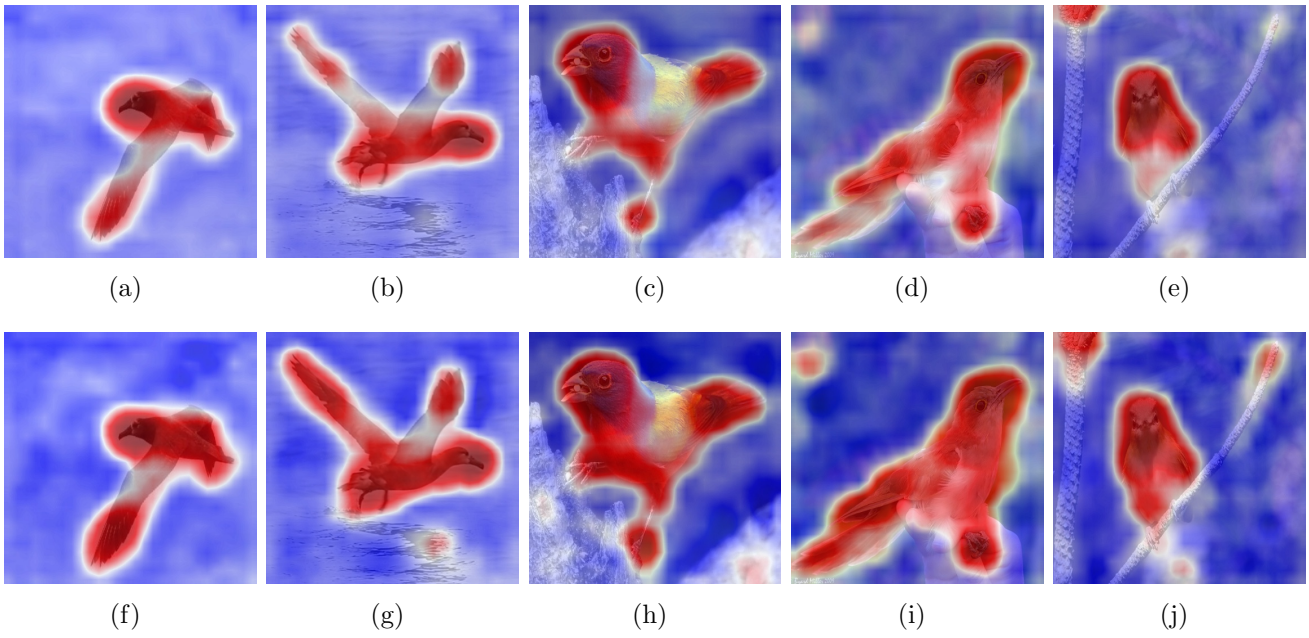

Figure 5: Visual comparison between the class activation maps generated by CCAM and CCNN. (a)-(e) are the results of CCAM, and the (f)-(j) is the results of CCNN.

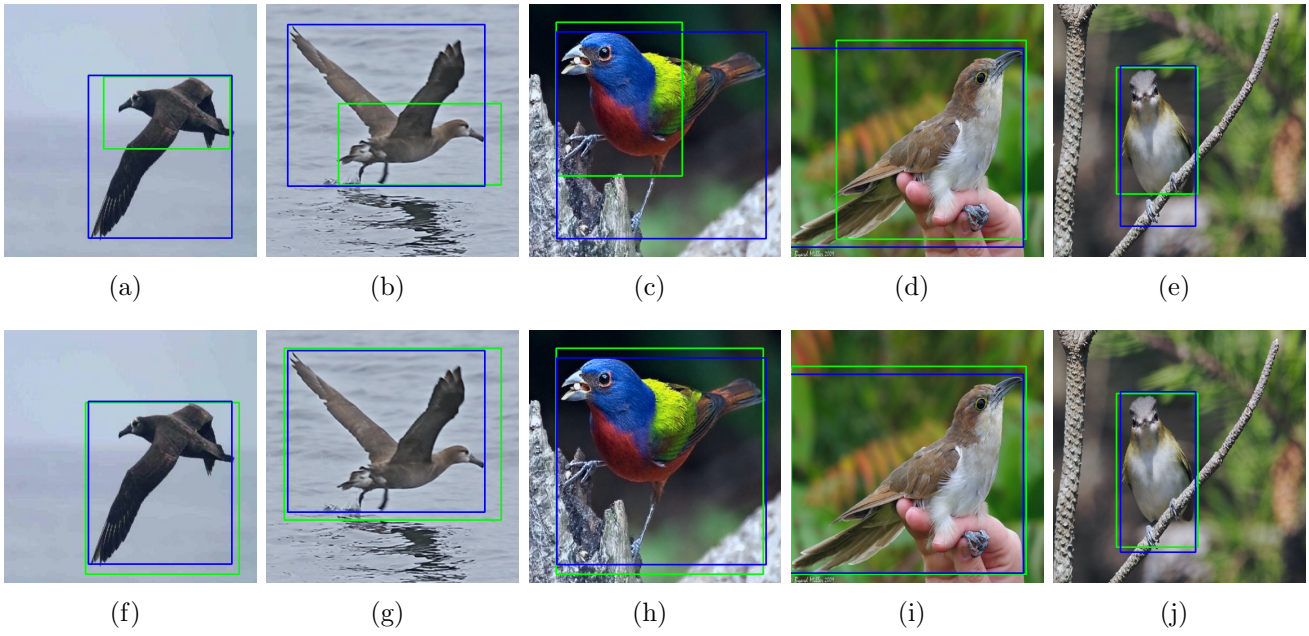

Figure 6: Visual comparison between the bounding boxes generated by CCAM and CCNN. (a)-(e) are the results of CCAM, and the (f)-(j) is the results of CCNN.

5.3 Rareults of WSSS

As this section is designed to demonstrate the effectivess of CAMs generated by the CCNN method, and that CAMs can be further used in other stages of WSSS. We provide a visualization of foreground objects in Figure 7. It can be observed that the activated objects are highly adhesive in the second column. However, the foreground objects activated by our method has good integrity. Therefore, we believe that the CAMs extracted by our model is helpful for downstream semantic segmentation tasks.

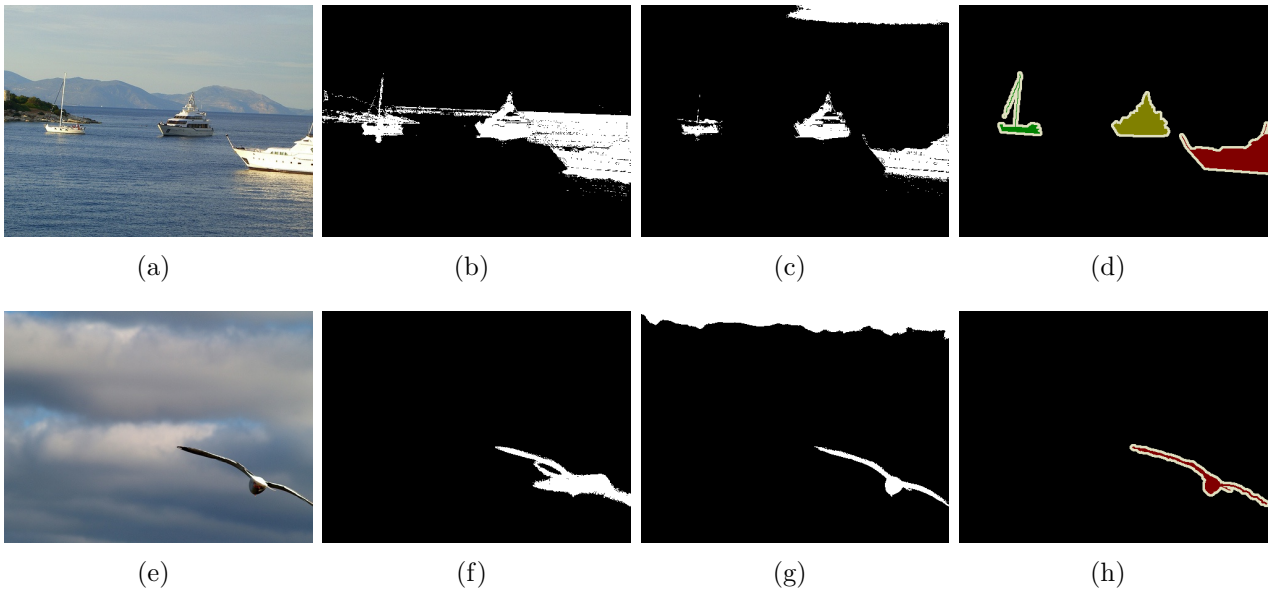

Figure 7: Visual comparison of the foreground objects segmented by CCAM and CCNN. The first column is original images. The second column is foreground target maps generated by CCAM. The third column is foreground target maps generated by CCNN. The last column is ground-truth masks.

6 Conclusion

In this report, we re-implement the code of the CCAM. We found that the backbone of the CCAM, which is the traditional convolutional neural networks ResNet50 that only activates the most discriminative regions. Therefore, we propose the CCNN method to solve the problem that the convolutional neural network and improve the performance of the CCAM algorithm. We also implement the python code of our CCNN model by referring to the code of the CCAM. Some experimental results suggest that our improved approach CCNN has a certain performance improvement compared to CCAM method.

References

[1] J. Dai, K. He, and J. Sun, "Boxsup: Exploiting bounding boxes to supervise convolutional networks for semantic segmentation," in *2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 1635–1643, 2015.

[2] D. Lin, J. Dai, J. Jia, K. He, and J. Sun, "Scribblesup: Scribble-supervised convolutional networks for semantic segmentation," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 3159–3167, 2016.

[3] A. Bearman, O. Russakovsky, V. Ferrari, and L. Fei-Fei, "What's the point: Semantic segmentation with point supervision," in *European conference on computer vision*, pp. 549–565, Springer, 2016.

[4] Y.-T. Chang, Q. Wang, W.-C. Hung, R. Piramuthu, Y.-H. Tsai, and M.-H. Yang, "Weakly-supervised semantic segmentation via sub-category exploration," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 8991–9000, 2020.

[5] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, and A. Torralba, "Learning deep features for discriminative localization," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2921–2929, 2016.

[6] J. Qin, J. Wu, X. Xiao, L. Li, and X. Wang, "Activation modulation and recalibration scheme for weakly supervised semantic segmentation," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 36, pp. 2117–2125, 2022.

[7] J. Xie, J. Xiang, J. Chen, X. Hou, X. Zhao, and L. Shen, "C2 am: Contrastive learning of class-agnostic activation map for weakly supervised object localization and semantic segmentation," in *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 979–988, 2022.

[8] C.-L. Zhang, Y.-H. Cao, and J. Wu, "Rethinking the route towards weakly supervised object localization," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 13460–13469, 2020.

[9] M. Tan and Q. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *International conference on machine learning*, pp. 6105–6114, PMLR, 2019.

[10] C. Wah, S. Branson, P. Welinder, P. Perona, and S. Belongie, "The caltech-ucsd birds-200-2011 dataset," 2011.

[11] M. Everingham and J. Winn, "The pascal visual object classes challenge 2012 (voc2012) development kit," *Pattern Anal. Stat. Model. Comput. Learn., Tech. Rep*, vol. 2007, pp. 1–45, 2012.

[12] J. Choe, S. J. Oh, S. Lee, S. Chun, Z. Akata, and H. Shim, "Evaluating weakly supervised object localization methods right," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 3133–3142, 2020.