卷积神经网络中的即插即用模块

GiantPandaCV 公众号出品

GiantPandaCV-pprp

2020-05-27

Contents

卷积神经网络中的即插即用模块

0. 序言

GiantPandaCV 公众号由专注于技术的一群 90 后创建, 专注于机器学习、深度学习、计算机视觉、图像处理等领域。半年以来已更新 **242** 篇原创技术文章。我们编写了《从零开始学习 **YOLOv3**》、《从零开始学习 **SSD**》、《**Faster R-CNN** 原理和代码讲解》、《多目标

跟踪快速入门》等系列原创电子书，关注后回复对应关键字即可免费领取。每天更新一到两篇相关推文, 希望在传播知识、分享知识的同时能够启发你。欢迎扫描下方二维码关注我们的公众号。

1. 即插即用模块简介

即插即用模块一般是作为一个独立的模块，可以用于取代普通的卷积结构，或者直接插入网络结构中。

最常见的即插即用模块莫过于注意力模块了，近些年好多略显水的工作都用到了注意力模块，仅仅需要简单添加这些注意力模块即可作为论文的创新点，比如 SENet+Darknet53 组合。

虽然笔者觉得这些模块有些真的是用来水文章的，但是不可否认很多模块确实增强了模型的特征表达能力。具体使用和评判还需要各位在自己的实验中以批判的眼光看待。

《卷积神经网络中的即插即用模块》电子书中，笔者将对自己接触过的即插即用模块进行简单讲解，不涉及核心，如果对具体设计思路和原理感兴趣，可以去公众号找对应的文章或者直接看对应的论文。

这里的即插即用模块主要分为注意力模块和其他模块。由于笔者本身涉猎有限，不可能将所有的即插即用模块都总结进来，所以如果有补充的可以联系笔者（微信名片在下），笔者在空闲时间将不断维护这个即插即用模块的项目。

项 目 地 址： https://github.com/pprp/SimpleCVReproduction/tree/master/Plug-and-play%20module

Figure 1: 笔者微信

一般来说，我们都很喜欢使用即插即用模块，因为其便于实现，可以快速验证，YOLOv4 中就提到了大量的即插即用模块。不过这些即插即用模块不一定对所有的任务都有效，笔者和一些群友交流过注意力模块方面的实验，在 YOLOv3 上通常可以带来 1 个百分点左右的提升，但是更多情况下是没有任何提升。添加这类即插即用模块还需要注意几个问题：

- 插入的位置：有的模块适合插入在浅层，有的模块适合在深层。具体插在哪里最好看原作者论文中的插入位置作为参考。一般情况可以插入的常见位置有：

  1. 瓶颈层：比如 ResNet，DenseNet 的瓶颈层。
  2. 上采样层：比如 FPN 分支，Attention UNet。
  3. 骨干网络最后一层：比如 SPP, ASPP 等
  4. 所有的 3x3 卷积：比如深度可分离卷积等

- 插入后进行实验为何不生效？指标没有提高甚至降低？

  很多模块虽然说是即插即用，但是并不是无脑插入以后结果就一定会提高。比如说，一个模型感受野已经很大，这时候如果在网络的浅层添加一些用于扩大感受野的模块，那样对结果不但不会有好的用处而且还会带来副作用。正确做法是，分析你网络的需要，根据需求选择对应功能的模块在合适的位置进行插入, 如果没有明确合适的位置，那就需要通过实验进行分析，确定哪个位置效果更佳（IBN-Net 中就是对不同位置使用 IBN 的结果进行了分析，最终确定了几种合适的方案）。

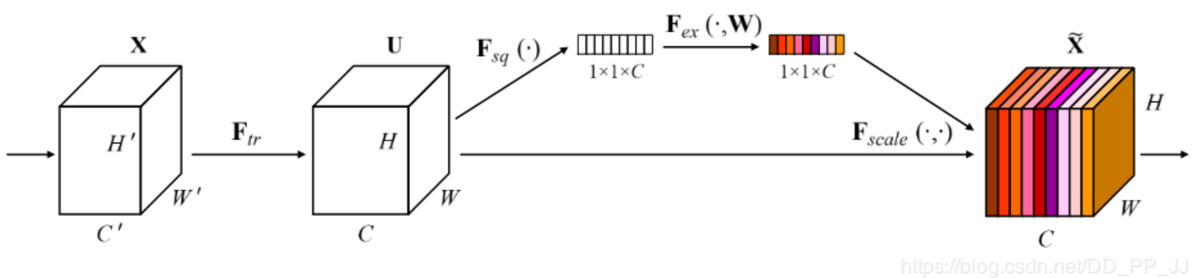
  另外，通过和几位知友的讨论，得知这些注意力模块通常情况下都需要调参才能维持原本的准确率，在调参效果比较好的情况下才能超过原本的模型。

- 即插即用模块的作用（以下内容的一个到多个）：

  1. 扩大模型感受野。
  2. 加快计算速度。
  3. 增加长距离依赖关系。
  4. 增加模型容量 (参数量增加了一部分)
  5. 提升了模型特征表达的多样性。

2. 注意力模块

2.1 SENet



说明：最经典的通道注意力模块，曾夺最后一节 ImageNet 冠军。

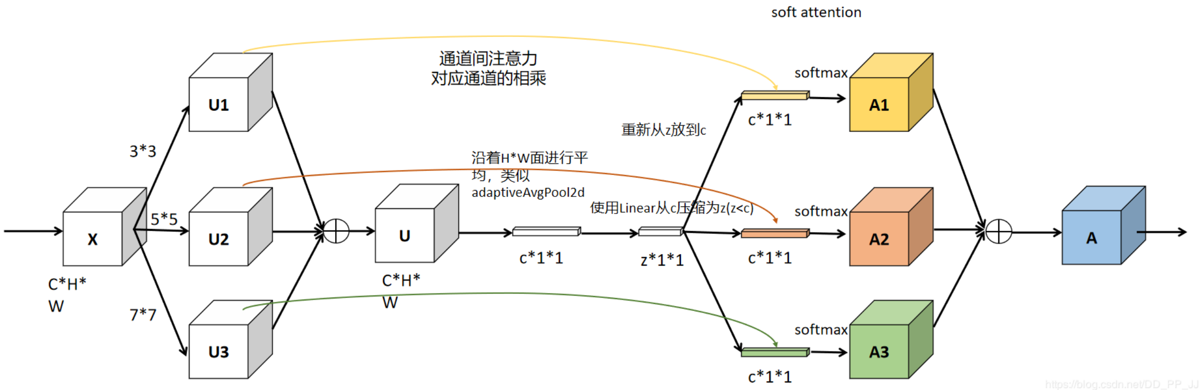
论文：https://arxiv.org/pdf/1709.01507

代码：

```python
import torch.nn as nn

class SELayer(nn.Module):
    def __init__(self, channel, reduction=16):
        super(SELayer, self).__init__()
        self.avgpool = nn.AdaptiveAvgPool2d(1)
        self.fc = nn.Sequential(
            nn.Linear(channel, channel//reduction,bias=False),
            nn.ReLU(inplace=True),
            nn.Linear(channel//reduction,channel, bias=False),
            nn.Sigmoid()
        )

    def forward(self, x):
        b,c,h,w = x.size()
        y = self.avgpool(x).view(b,c)
```

```
        y = self.fc(y).view(b,c,1,1)
        return x * y.expand_as(x)
```

2.2 SKNet



说明：SENet 改进版，增加了多个分支，每个分支感受野不同。

论文：https://arxiv.org/pdf/1903.06586

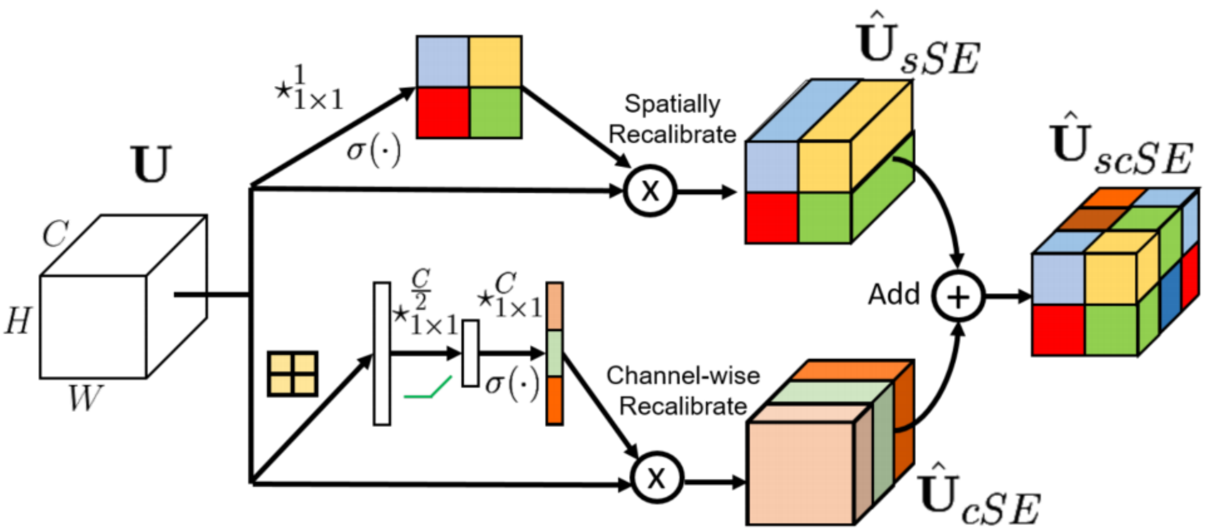
代码：

```
import torch.nn as nn
import torch

class SKConv(nn.Module):
    def __init__(self, features, WH, M, G, r, stride=1, L=32):
        """ Constructor
        Args:
            features: input channel dimensionality.
            WH: input spatial dimensionality, used for GAP kernel size.
            M: the number of branchs.
            G: num of convolution groups.
            r: the radio for compute d, the length of z.
            stride: stride, default 1.
            L: the minimum dim of the vector z in paper, default 32.
        """
        super(SKConv, self).__init__()
        d = max(int(features / r), L)
        self.M = M
        self.features = features
        self.convs = nn.ModuleList([])
        for i in range(M):
```

```
            self.convs.append(
                nn.Sequential(
                    nn.Conv2d(features,
                              features,
                              kernel_size=3 + i * 2,
                              stride=stride,
                              padding=1 + i,
                              groups=G), nn.BatchNorm2d(features),
                    nn.ReLU(inplace=False)))
        # self.gap = nn.AvgPool2d(int(WH/stride))
        print("D:", d)
        self.fc = nn.Linear(features, d)
        self.fcs = nn.ModuleList([])
        for i in range(M):
            self.fcs.append(nn.Linear(d, features))
        self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        for i, conv in enumerate(self.convs):
            fea = conv(x).unsqueeze_(dim=1)
            if i == 0:
                feas = fea
            else:
                feas = torch.cat([feas, fea], dim=1)
        fea_U = torch.sum(feas, dim=1)
        fea_s = fea_U.mean(-1).mean(-1)
        fea_z = self.fc(fea_s)
        for i, fc in enumerate(self.fcs):
            print(i, fea_z.shape)
            vector = fc(fea_z).unsqueeze_(dim=1)
            print(i, vector.shape)
            if i == 0:
                attention_vectors = vector
            else:
                attention_vectors = torch.cat([attention_vectors, vector],
                                              dim=1)
        attention_vectors = self.softmax(attention_vectors)
        attention_vectors = attention_vectors.unsqueeze(-1).unsqueeze(-1)
        fea_v = (feas * attention_vectors).sum(dim=1)
        return fea_v

if __name__ == "__main__":
    t = torch.ones((32, 256, 24,24))
```

```
    sk = SKConv(256,WH=1,M=2,G=1,r=2)
    out = sk(t)
    print(out.shape)
```

2.3 scSE



(d) Concurrent Spatial and Channel Squeeze and Channel Excitation (scSE)

说明：scSE 分为两个模块，一个是 sSE 和 cSE 模块，分别是空间注意力和通道注意力，最终以相加的方式融合。论文中只将其使用在分割模型中，在很多图像分割比赛中都有用到这个模块作为 trick。

论文：http://arxiv.org/pdf/1803.02579v2

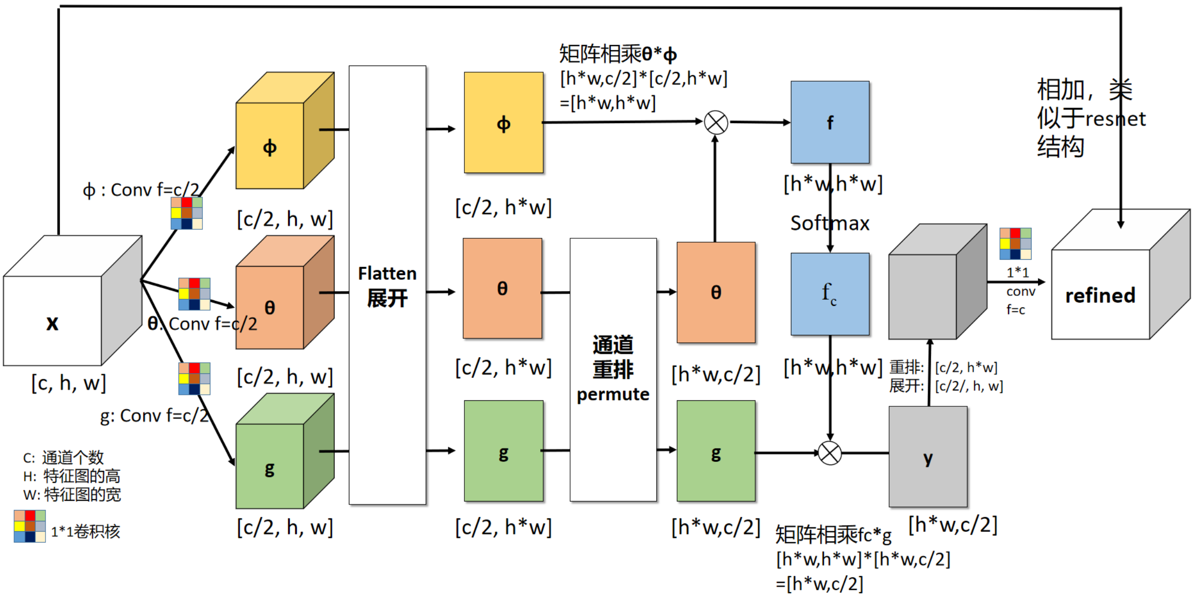
代码：

```
import torch
import torch.nn as nn


class sSE(nn.Module):
    def __init__(self, in_channels):
        super().__init__()
        self.Conv1x1 = nn.Conv2d(in_channels, 1, kernel_size=1, bias=False)
        self.norm = nn.Sigmoid()

    def forward(self, U):
        q = self.Conv1x1(U)  # U:[bs,c,h,w] to q:[bs,1,h,w]
```

```
        q = self.norm(q)
        return U * q  # 广播机制

class cSE(nn.Module):
    def __init__(self, in_channels):
        super().__init__()
        self.avgpool = nn.AdaptiveAvgPool2d(1)
        self.Conv_Squeeze = nn.Conv2d(in_channels, in_channels // 2,
        ↪  kernel_size=1, bias=False)
        self.Conv_Excitation = nn.Conv2d(in_channels//2, in_channels,
        ↪  kernel_size=1, bias=False)
        self.norm = nn.Sigmoid()

    def forward(self, U):
        z = self.avgpool(U)# shape: [bs, c, h, w] to [bs, c, 1, 1]
        z = self.Conv_Squeeze(z) # shape: [bs, c/2]
        z = self.Conv_Excitation(z) # shape: [bs, c]
        z = self.norm(z)
        return U * z.expand_as(U)

class scSE(nn.Module):
    def __init__(self, in_channels):
        super().__init__()
        self.cSE = cSE(in_channels)
        self.sSE = sSE(in_channels)

    def forward(self, U):
        U_sse = self.sSE(U)
        U_cse = self.cSE(U)
        return U_cse+U_sse

if __name__ == "__main__":
    bs, c, h, w = 10, 3, 64, 64
    in_tensor = torch.ones(bs, c, h, w)

    sc_se = scSE(c)
    print("in shape:",in_tensor.shape)
    out_tensor = sc_se(in_tensor)
    print("out shape:", out_tensor.shape)
```

2.4 Non-Local Net

$$y_i = \mathrm{soft\,max}(\theta(x_i)^T \phi(x_j))g(x_j) = \frac{1}{\sum_{\forall j} e^{\theta(x_i)^T \phi(x_j)}} e^{\theta(x_i)^T \phi(x_j)} W_g x_j$$



说明：NLNet 主要借鉴了传统方法中的非局部均值滤波设计了 Non-Local 全局注意力，虽然效果好，但是计算量偏大，建议不要在底层网络使用，可以适当在高层网络中使用。

论文：https://arxiv.org/pdf/1711.07971

代码：

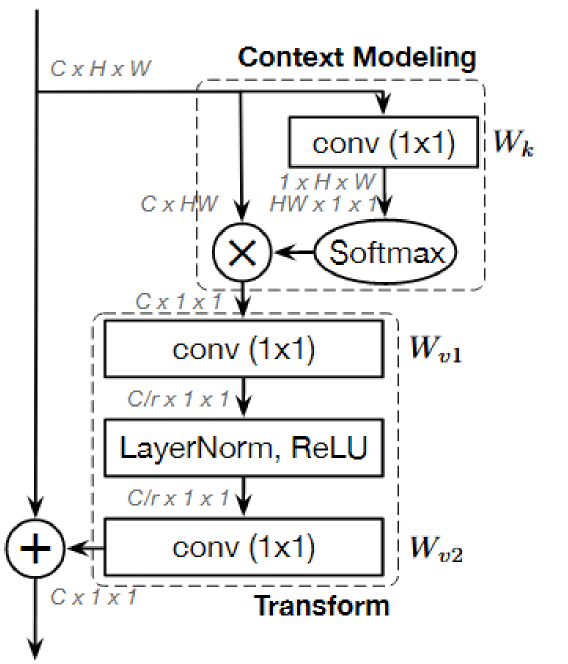
```python
import torch
from torch import nn
from torch.nn import functional as F


class _NonLocalBlockND(nn.Module):
    """
    调用过程
    NONLocalBlock2D(in_channels=32),
    super(NONLocalBlock2D, self).__init__(in_channels,
            inter_channels=inter_channels,
            dimension=2, sub_sample=sub_sample,
```

```
            bn_layer=bn_layer)
    """
    def __init__(self,
                 in_channels,
                 inter_channels=None,
                 dimension=3,
                 sub_sample=True,
                 bn_layer=True):
        super(_NonLocalBlockND, self).__init__()

        assert dimension in [1, 2, 3]

        self.dimension = dimension
        self.sub_sample = sub_sample

        self.in_channels = in_channels
        self.inter_channels = inter_channels

        if self.inter_channels is None:
            self.inter_channels = in_channels // 2
            # 进行压缩得到 channel 个数
            if self.inter_channels == 0:
                self.inter_channels = 1

        if dimension == 3:
            conv_nd = nn.Conv3d
            max_pool_layer = nn.MaxPool3d(kernel_size=(1, 2, 2))
            bn = nn.BatchNorm3d
        elif dimension == 2:
            conv_nd = nn.Conv2d
            max_pool_layer = nn.MaxPool2d(kernel_size=(2, 2))
            bn = nn.BatchNorm2d
        else:
            conv_nd = nn.Conv1d
            max_pool_layer = nn.MaxPool1d(kernel_size=(2))
            bn = nn.BatchNorm1d

        self.g = conv_nd(in_channels=self.in_channels,
                         out_channels=self.inter_channels,
                         kernel_size=1,
                         stride=1,
                         padding=0)
```

```python
        if bn_layer:
            self.W = nn.Sequential(
                conv_nd(in_channels=self.inter_channels,
                        out_channels=self.in_channels,
                        kernel_size=1,
                        stride=1,
                        padding=0), bn(self.in_channels))
            nn.init.constant_(self.W[1].weight, 0)
            nn.init.constant_(self.W[1].bias, 0)
        else:
            self.W = conv_nd(in_channels=self.inter_channels,
                             out_channels=self.in_channels,
                             kernel_size=1,
                             stride=1,
                             padding=0)
            nn.init.constant_(self.W.weight, 0)
            nn.init.constant_(self.W.bias, 0)

        self.theta = conv_nd(in_channels=self.in_channels,
                             out_channels=self.inter_channels,
                             kernel_size=1,
                             stride=1,
                             padding=0)
        self.phi = conv_nd(in_channels=self.in_channels,
                           out_channels=self.inter_channels,
                           kernel_size=1,
                           stride=1,
                           padding=0)

        if sub_sample:
            self.g = nn.Sequential(self.g, max_pool_layer)
            self.phi = nn.Sequential(self.phi, max_pool_layer)

    def forward(self, x):
        '''
        :param x: (b, c,  h, w)
        :return:
        '''

        batch_size = x.size(0)

        g_x = self.g(x).view(batch_size, self.inter_channels, -1)#[bs, c,
↪   w*h]
```

```
g_x = g_x.permute(0, 2, 1)

theta_x = self.theta(x).view(batch_size, self.inter_channels, -1)
theta_x = theta_x.permute(0, 2, 1)

phi_x = self.phi(x).view(batch_size, self.inter_channels, -1)

f = torch.matmul(theta_x, phi_x)

print(f.shape)

f_div_C = F.softmax(f, dim=-1)

y = torch.matmul(f_div_C, g_x)
y = y.permute(0, 2, 1).contiguous()
y = y.view(batch_size, self.inter_channels, *x.size()[2:])
W_y = self.W(y)
z = W_y + x
return z
```

2.5 GCNet



(d) Global context (GC) block

说明：GCNet 主要针对 Non-Local 计算量过大的问题结合了提出了解决方案

论文：https://arxiv.org/abs/1904.11492

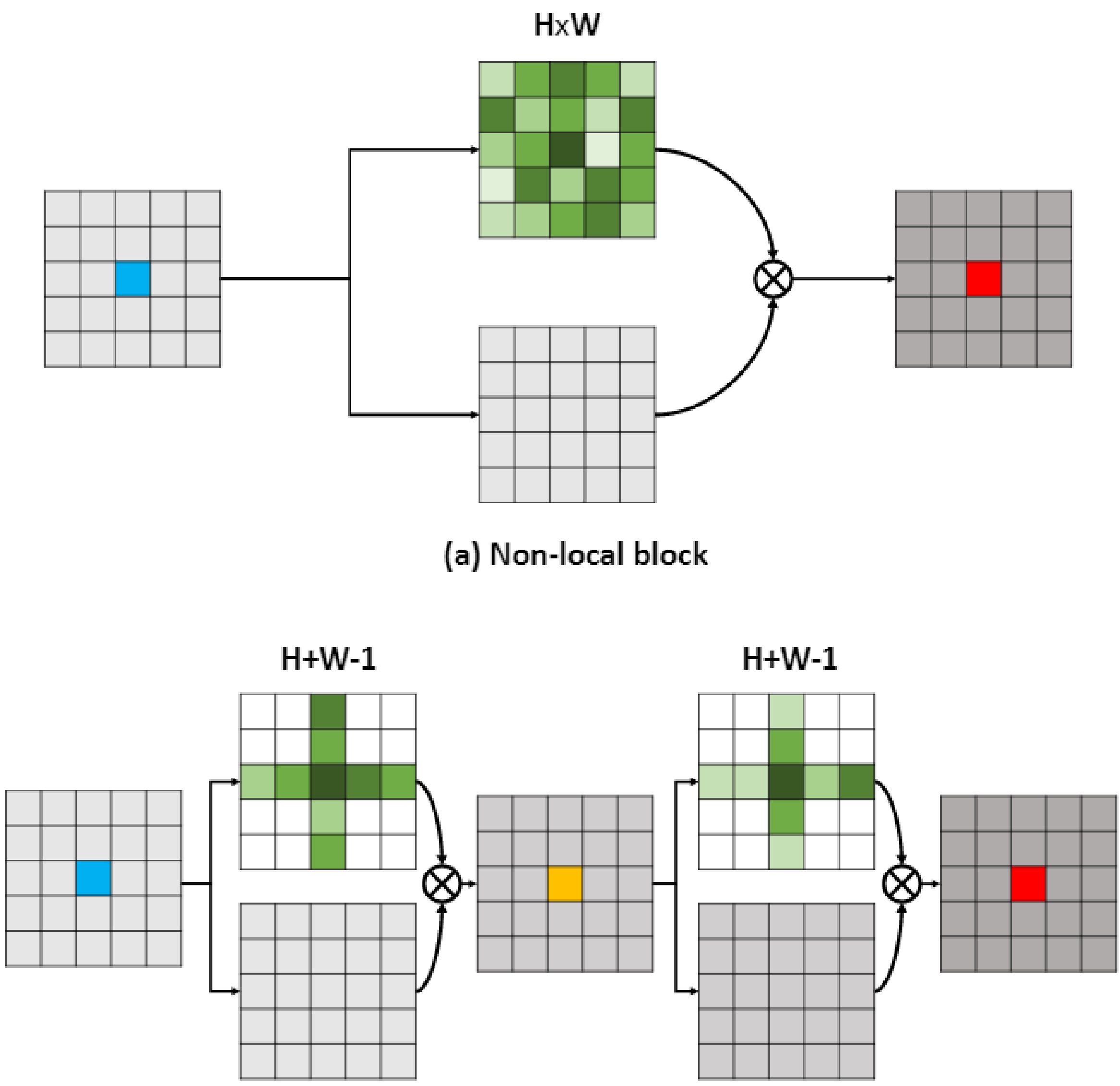
代码：

```
import torch
from torch import nn

class ContextBlock(nn.Module):
    def __init__(self,inplanes,ratio,pooling_type='att',
                 fusion_types=('channel_add', )):
        super(ContextBlock, self).__init__()
        valid_fusion_types = ['channel_add', 'channel_mul']

        assert pooling_type in ['avg', 'att']
        assert isinstance(fusion_types, (list, tuple))
        assert all([f in valid_fusion_types for f in fusion_types])
        assert len(fusion_types) > 0, 'at least one fusion should be used'

        self.inplanes = inplanes
        self.ratio = ratio
        self.planes = int(inplanes * ratio)
        self.pooling_type = pooling_type
        self.fusion_types = fusion_types

        if pooling_type == 'att':
            self.conv_mask = nn.Conv2d(inplanes, 1, kernel_size=1)
            self.softmax = nn.Softmax(dim=2)
        else:
            self.avg_pool = nn.AdaptiveAvgPool2d(1)
        if 'channel_add' in fusion_types:
            self.channel_add_conv = nn.Sequential(
                nn.Conv2d(self.inplanes, self.planes, kernel_size=1),
                nn.LayerNorm([self.planes, 1, 1]),
                nn.ReLU(inplace=True),  # yapf: disable
                nn.Conv2d(self.planes, self.inplanes, kernel_size=1))
        else:
            self.channel_add_conv = None
        if 'channel_mul' in fusion_types:
            self.channel_mul_conv = nn.Sequential(
                nn.Conv2d(self.inplanes, self.planes, kernel_size=1),
                nn.LayerNorm([self.planes, 1, 1]),
                nn.ReLU(inplace=True),  # yapf: disable
```

```
            nn.Conv2d(self.planes, self.inplanes, kernel_size=1))
    else:
        self.channel_mul_conv = None


def spatial_pool(self, x):
    batch, channel, height, width = x.size()
    if self.pooling_type == 'att':
        input_x = x
        # [N, C, H * W]
        input_x = input_x.view(batch, channel, height * width)
        # [N, 1, C, H * W]
        input_x = input_x.unsqueeze(1)
        # [N, 1, H, W]
        context_mask = self.conv_mask(x)
        # [N, 1, H * W]
        context_mask = context_mask.view(batch, 1, height * width)
        # [N, 1, H * W]
        context_mask = self.softmax(context_mask)
        # [N, 1, H * W, 1]
        context_mask = context_mask.unsqueeze(-1)
        # [N, 1, C, 1]
        context = torch.matmul(input_x, context_mask)
        # [N, C, 1, 1]
        context = context.view(batch, channel, 1, 1)
    else:
        # [N, C, 1, 1]
        context = self.avg_pool(x)
    return context

def forward(self, x):
    # [N, C, 1, 1]
    context = self.spatial_pool(x)
    out = x
    if self.channel_mul_conv is not None:
        # [N, C, 1, 1]
        channel_mul_term = torch.sigmoid(self.channel_mul_conv(context))
        out = out * channel_mul_term
    if self.channel_add_conv is not None:
        # [N, C, 1, 1]
        channel_add_term = self.channel_add_conv(context)
        out = out + channel_add_term
    return out
```

```python
if __name__ == "__main__":
    in_tensor = torch.ones((12, 64, 128, 128))

    cb = ContextBlock(inplanes=64, ratio=1./16.,pooling_type='att')

    out_tensor = cb(in_tensor)

    print(in_tensor.shape)
    print(out_tensor.shape)
```

2.6 CCNet



(a) Non-local block

(b) Criss-Cross Attention block

说明：也是 Non-Local 发展而来的注意力模块，其特殊之处在纵横交叉关注模块，可以以更有效的方式从远程依赖中获取上下文信息。

论文：https://arxiv.org/abs/1811.11721

代码：https://github.com/speedinghzl/CCNet

```
class CrissCrossAttention(nn.Module):
```

```
    """ Criss-Cross Attention Module"""
    def __init__(self, in_dim):
        super(CrissCrossAttention, self).__init__()
        self.chanel_in = in_dim

        self.query_conv = nn.Conv2d(in_channels=in_dim,
                                    out_channels=in_dim // 8,
                                    kernel_size=1)
        self.key_conv = nn.Conv2d(in_channels=in_dim,
                                  out_channels=in_dim // 8,
                                  kernel_size=1)
        self.value_conv = nn.Conv2d(in_channels=in_dim,
                                    out_channels=in_dim,
                                    kernel_size=1)
        self.gamma = nn.Parameter(torch.zeros(1))

    def forward(self, x):
        proj_query = self.query_conv(x)
        proj_key = self.key_conv(x)
        proj_value = self.value_conv(x)

        energy = ca_weight(proj_query, proj_key)
        attention = F.softmax(energy, 1)
        out = ca_map(attention, proj_value)
        out = self.gamma * out + x

        return out
```

2.7 CBAM

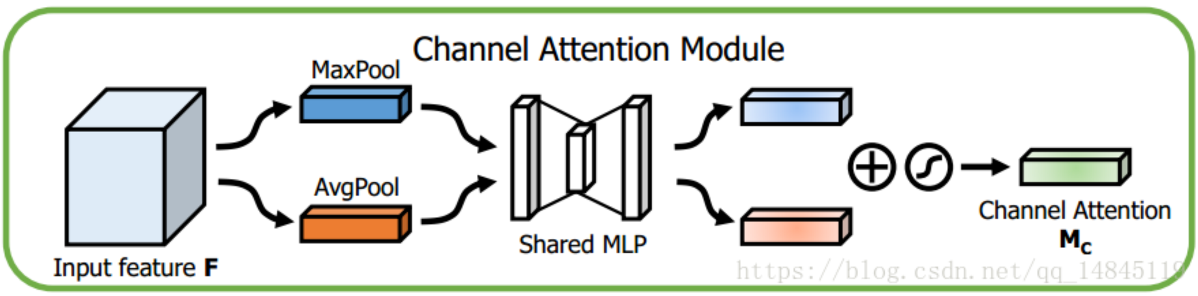


Figure 2: 通道注意力

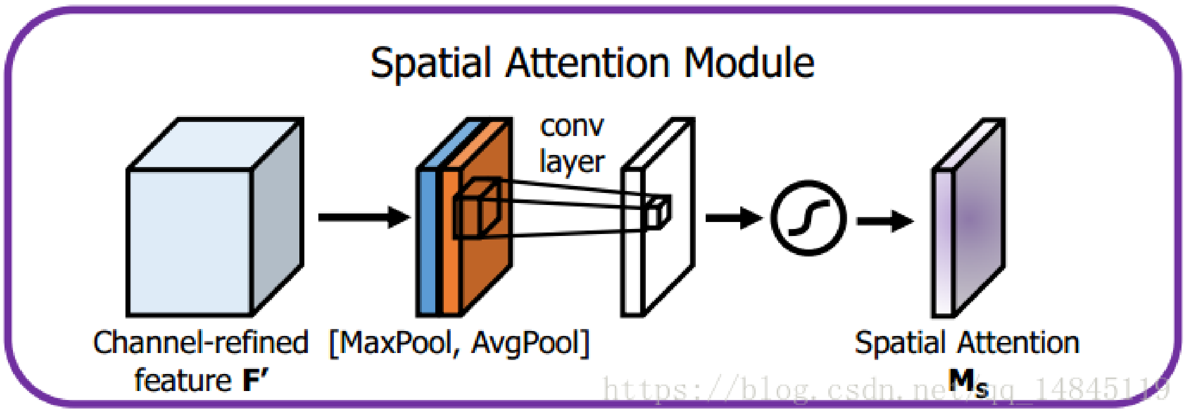


Figure 3: 空间注意力

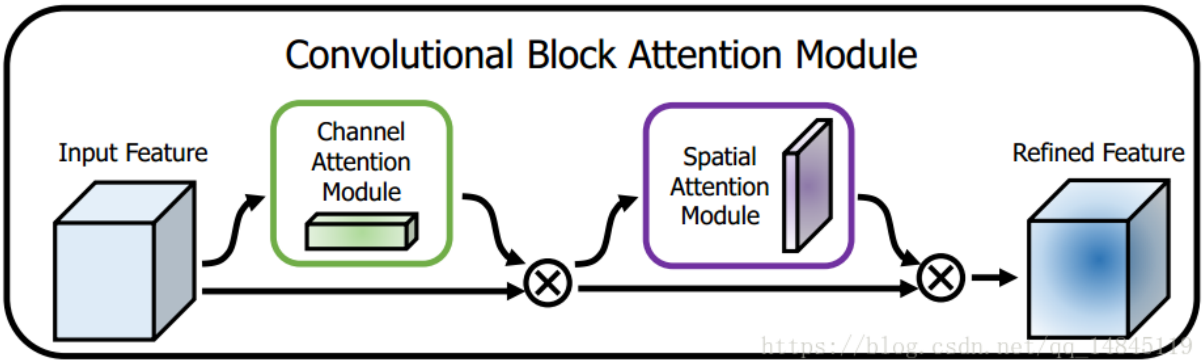


Figure 4: CBAM

说明：将空间注意力机制和通道注意力机制进行串联

论文：https://arxiv.org/abs/1807.06521

代码：

```
import torch
import torch.nn as nn


def conv3x3(in_planes, out_planes, stride=1):
    "3x3 convolution with padding"
    return nn.Conv2d(in_planes,
                     out_planes,
                     kernel_size=3,
                     stride=stride,
```

```
                    padding=1,
                    bias=False)


class ChannelAttention(nn.Module):
    def __init__(self, in_planes, ratio=4):
        super(ChannelAttention, self).__init__()
        self.avg_pool = nn.AdaptiveAvgPool2d(1)
        self.max_pool = nn.AdaptiveMaxPool2d(1)

        self.sharedMLP = nn.Sequential(
            nn.Conv2d(in_planes, in_planes // ratio, 1, bias=False),
 ↪  nn.ReLU(),
            nn.Conv2d(in_planes // ratio, in_planes, 1, bias=False))
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        avgout = self.sharedMLP(self.avg_pool(x))
        maxout = self.sharedMLP(self.max_pool(x))
        return self.sigmoid(avgout + maxout)


class SpatialAttention(nn.Module):
    def __init__(self, kernel_size=7):
        super(SpatialAttention, self).__init__()
        assert kernel_size in (3, 7), "kernel size must be 3 or 7"
        padding = 3 if kernel_size == 7 else 1

       self.conv = nn.Conv2d(2, 1, kernel_size, padding=padding, bias=False)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        avgout = torch.mean(x, dim=1, keepdim=True)
        maxout, _ = torch.max(x, dim=1, keepdim=True)
        x = torch.cat([avgout, maxout], dim=1)
        x = self.conv(x)
        return self.sigmoid(x)


class BasicBlock(nn.Module):
    expansion = 1

    def __init__(self, inplanes, planes, stride=1, downsample=None):
```

```
        super(BasicBlock, self).__init__()
        self.conv1 = conv3x3(inplanes, planes, stride)
        self.bn1 = nn.BatchNorm2d(planes)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(planes, planes)
        self.bn2 = nn.BatchNorm2d(planes)

        self.ca = ChannelAttention(planes)
        self.sa = SpatialAttention()

        self.downsample = downsample
        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.ca(out) * out  # 广播机制
        out = self.sa(out) * out  # 广播机制

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

        print(out.shape, residual.shape)

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

        return out


if __name__ == "__main__":
    downsample = nn.Sequential(
        nn.Conv2d(16, 32, kernel_size=1, stride=1, bias=False),
        nn.BatchNorm2d(32))
```

```
    x = torch.ones(3, 16, 32, 32)

    model = BasicBlock(16, 32, stride=1, downsample=downsample)

    print(model(x).shape)
```

2.8 BAM

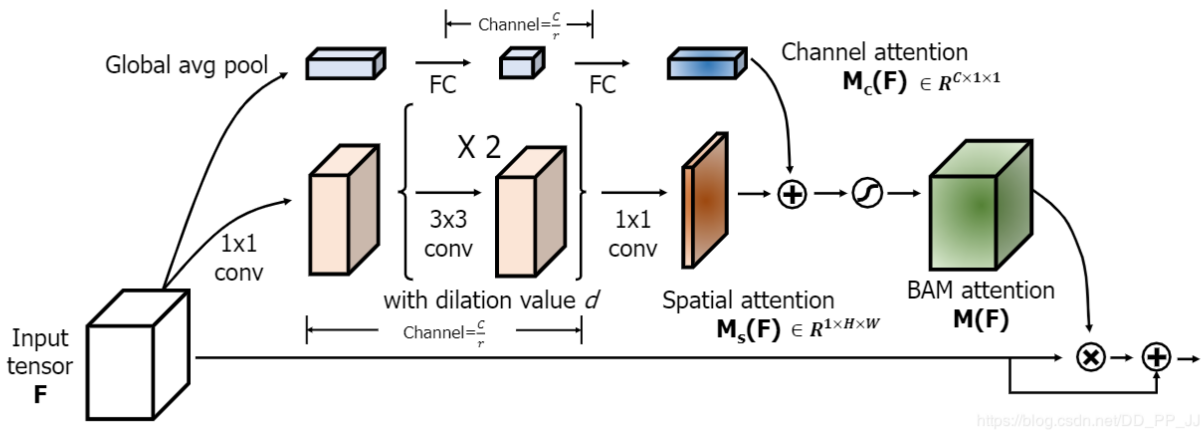


Figure 5: BAM

说明：和 CBAM 同一个作者，将通道注意力和空间注意力用并联的方式连接

论文：https://arxiv.org/abs/1807.06514

代码：

```
import torch
import math
import torch.nn as nn
import torch.nn.functional as F


class Flatten(nn.Module):
    def forward(self, x):
        return x.view(x.size(0), -1)


class ChannelGate(nn.Module):
    def __init__(self, gate_channel, reduction_ratio=16, num_layers=1):
        super(ChannelGate, self).__init__()
```

```
        self.gate_c = nn.Sequential()
        self.gate_c.add_module('flatten', Flatten())

        gate_channels = [gate_channel]  # eg 64
        gate_channels += [gate_channel // reduction_ratio] * num_layers  #
↪   eg 4
        gate_channels += [gate_channel]  # 64
        # gate_channels: [64, 4, 4]

        for i in range(len(gate_channels) - 2):
            self.gate_c.add_module(
                'gate_c_fc_%d' % i,
                nn.Linear(gate_channels[i], gate_channels[i + 1]))
            self.gate_c.add_module('gate_c_bn_%d' % (i + 1),
                                   nn.BatchNorm1d(gate_channels[i + 1]))
            self.gate_c.add_module('gate_c_relu_%d' % (i + 1), nn.ReLU())

        self.gate_c.add_module('gate_c_fc_final',
                               nn.Linear(gate_channels[-2], gate_channels[-1]))

    def forward(self, x):
        avg_pool = F.avg_pool2d(x, x.size(2), stride=x.size(2))
        return self.gate_c(avg_pool).unsqueeze(2).unsqueeze(3).expand_as(x)


class SpatialGate(nn.Module):
    def __init__(self,
                 gate_channel,
                 reduction_ratio=16,
                 dilation_conv_num=2,
                 dilation_val=4):
        super(SpatialGate, self).__init__()
        self.gate_s = nn.Sequential()

        self.gate_s.add_module(
            'gate_s_conv_reduce0',
            nn.Conv2d(gate_channel,
                      gate_channel // reduction_ratio,
                      kernel_size=1))
        self.gate_s.add_module('gate_s_bn_reduce0',
                               nn.BatchNorm2d(gate_channel // reduction_ratio))
        self.gate_s.add_module('gate_s_relu_reduce0', nn.ReLU())
```

```python
        # 进行多个空洞卷积，丰富感受野
        for i in range(dilation_conv_num):
            self.gate_s.add_module(
                'gate_s_conv_di_%d' % i,
                nn.Conv2d(gate_channel // reduction_ratio,
                          gate_channel // reduction_ratio,
                          kernel_size=3,
                          padding=dilation_val,
                          dilation=dilation_val))
            self.gate_s.add_module(
                'gate_s_bn_di_%d' % i,
                nn.BatchNorm2d(gate_channel // reduction_ratio))
            self.gate_s.add_module('gate_s_relu_di_%d' % i, nn.ReLU())

        self.gate_s.add_module(
            'gate_s_conv_final',
            nn.Conv2d(gate_channel // reduction_ratio, 1, kernel_size=1))

    def forward(self, x):
        return self.gate_s(x).expand_as(x)


class BAM(nn.Module):
    def __init__(self, gate_channel):
        super(BAM, self).__init__()
        self.channel_att = ChannelGate(gate_channel)
        self.spatial_att = SpatialGate(gate_channel)

    def forward(self, x):
        att = 1 + F.sigmoid(self.channel_att(x) * self.spatial_att(x))
        return att * x
```

2.9 SplitAttention

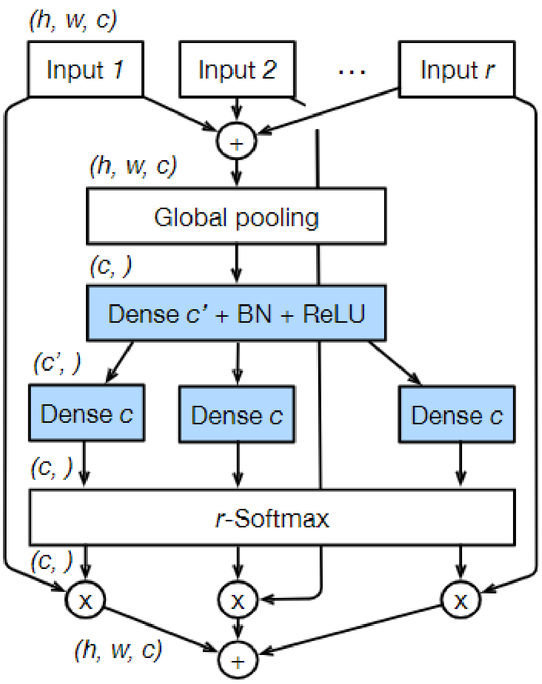


Fig. 2: Split-Attention within a cardinal group. For easy visualization in the figure, we use $c = C/K$ in this figure.

说明：ResNeSt = SENet + SKNet + ResNeXt

论文：https://hangzhang.org/files/resnest.pdf

代码：

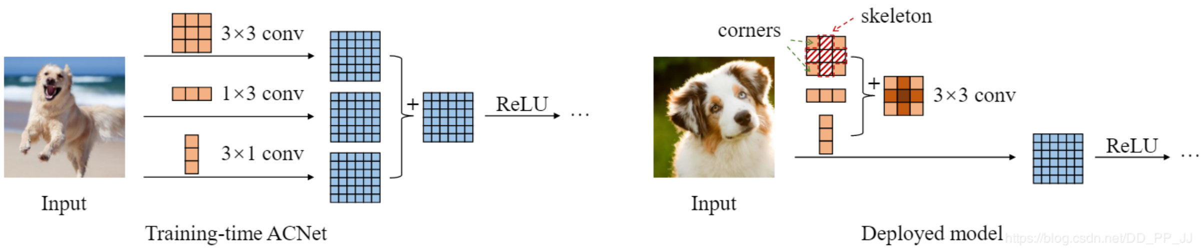
```
import torch
from torch import nn
import torch.nn.functional as F
from torch.nn import Conv2d, Module, Linear, BatchNorm2d, ReLU
from torch.nn.modules.utils import _pair

__all__ = ['SplAtConv2d']

class SplAtConv2d(Module):
    """Split-Attention Conv2d
    """
```

```
    def __init__(self, in_channels, channels, kernel_size, stride=(1, 1),
     ↪ padding=(0, 0),
                 dilation=(1, 1), groups=1, bias=True,
                 radix=2, reduction_factor=4,
                 rectify=False, rectify_avg=False, norm_layer=None,
                 dropblock_prob=0.0, **kwargs):
        super(SplAtConv2d, self).__init__()
        padding = _pair(padding)
        self.rectify = rectify and (padding[0] > 0 or padding[1] > 0)
        self.rectify_avg = rectify_avg
        inter_channels = max(in_channels*radix//reduction_factor, 32)
        self.radix = radix
        self.cardinality = groups
        self.channels = channels
        self.dropblock_prob = dropblock_prob
        if self.rectify:
            from rfconv import RFConv2d
            self.conv = RFConv2d(in_channels, channels*radix, kernel_size,
             ↪ stride, padding, dilation,
                                 groups=groups*radix, bias=bias,
↪   average_mode=rectify_avg, **kwargs)
        else:
            self.conv = Conv2d(in_channels, channels*radix, kernel_size,
             ↪ stride, padding, dilation,
                               groups=groups*radix, bias=bias, **kwargs)
        self.use_bn = norm_layer is not None
        if self.use_bn:
            self.bn0 = norm_layer(channels*radix)
        self.relu = ReLU(inplace=True)
        self.fc1 = Conv2d(channels, inter_channels, 1,
         ↪ groups=self.cardinality)
        if self.use_bn:
            self.bn1 = norm_layer(inter_channels)
        self.fc2 = Conv2d(inter_channels, channels*radix, 1,
         ↪ groups=self.cardinality)
        if dropblock_prob > 0.0:
            self.dropblock = DropBlock2D(dropblock_prob, 3)
        self.rsoftmax = rSoftMax(radix, groups)

    def forward(self, x):
        x = self.conv(x)
        if self.use_bn:
            x = self.bn0(x)
```

```python
        if self.dropblock_prob > 0.0:
            x = self.dropblock(x)
        x = self.relu(x)

        batch, rchannel = x.shape[:2]
        if self.radix > 1:
            splited = torch.split(x, rchannel//self.radix, dim=1)
            gap = sum(splited)
        else:
            gap = x
        gap = F.adaptive_avg_pool2d(gap, 1)
        gap = self.fc1(gap)

        if self.use_bn:
            gap = self.bn1(gap)
        gap = self.relu(gap)

        atten = self.fc2(gap)
        atten = self.rsoftmax(atten).view(batch, -1, 1, 1)

        if self.radix > 1:
            attens = torch.split(atten, rchannel//self.radix, dim=1)
            out = sum([att*split for (att, split) in zip(attens, splited)])
        else:
            out = atten * x
        return out.contiguous()

class rSoftMax(nn.Module):
    def __init__(self, radix, cardinality):
        super().__init__()
        self.radix = radix
        self.cardinality = cardinality

    def forward(self, x):
        batch = x.size(0)
        if self.radix > 1:
            x = x.view(batch, self.cardinality, self.radix, -1).transpose(1,
↪   2)
            x = F.softmax(x, dim=1)
            x = x.reshape(batch, -1)
        else:
            x = torch.sigmoid(x)
        return x
```

3. 其他模块

3.1 ACNet



说明：通过在训练过程中引入 1x3 conv 和 3x1 conv，强化特征提取，实现效果提升

论文：ACNet: Strengthening the Kernel Skeletons for Powerful CNN via Asymmetric Convolution Blocks.

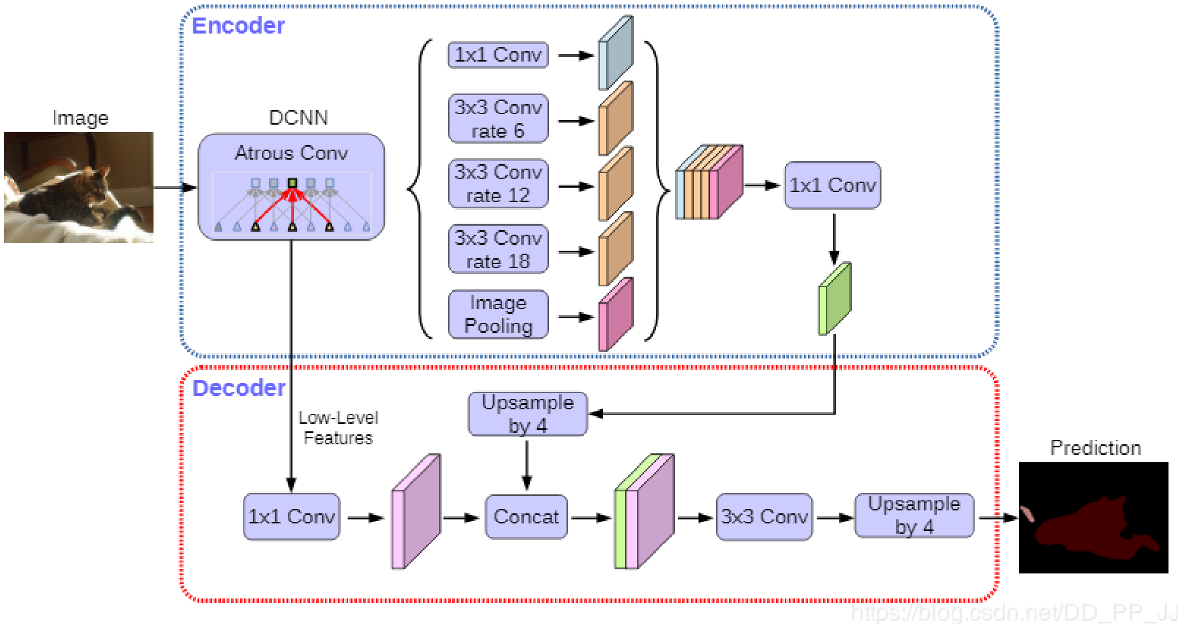
代码：

```
import torch.nn as nn
import torch


class CropLayer(nn.Module):

    #   E.g., (-1, 0) means this layer should crop the first and last rows of
    ↪  the feature map. And (0, -1) crops the first and last columns
    def __init__(self, crop_set):
        super(CropLayer, self).__init__()
        self.rows_to_crop = - crop_set[0]
        self.cols_to_crop = - crop_set[1]
        assert self.rows_to_crop >= 0
        assert self.cols_to_crop >= 0

    def forward(self, input):
        return input[:, :, self.rows_to_crop:-self.rows_to_crop,
         ↪  self.cols_to_crop:-self.cols_to_crop]


class ACBlock(nn.Module):
    def __init__(self,
                 in_channels,
                 out_channels,
                 kernel_size,
```

```
                 stride=1,
                 padding=0,
                 dilation=1,
                 groups=1,
                 padding_mode='zeros',
                 deploy=False):
        super(ACBlock, self).__init__()
        self.deploy = deploy
        if deploy:
            self.fused_conv = nn.Conv2d(in_channels=in_channels,
                                        out_channels=out_channels,
                                     kernel_size=(kernel_size, kernel_size),
                                        stride=stride,
                                        padding=padding,
                                        dilation=dilation,
                                        groups=groups,
                                        bias=True,
                                        padding_mode=padding_mode)
        else:
            self.square_conv = nn.Conv2d(in_channels=in_channels,
                                         out_channels=out_channels,
                                         kernel_size=(kernel_size,
                                                      kernel_size),
                                         stride=stride,
                                         padding=padding,
                                         dilation=dilation,
                                         groups=groups,
                                         bias=False,
                                         padding_mode=padding_mode)
            self.square_bn = nn.BatchNorm2d(num_features=out_channels)

            center_offset_from_origin_border = padding - kernel_size // 2
            ver_pad_or_crop = (center_offset_from_origin_border + 1,
                               center_offset_from_origin_border)
            hor_pad_or_crop = (center_offset_from_origin_border,
                               center_offset_from_origin_border + 1)
            if center_offset_from_origin_border >= 0:
                self.ver_conv_crop_layer = nn.Identity()
                ver_conv_padding = ver_pad_or_crop
                self.hor_conv_crop_layer = nn.Identity()
                hor_conv_padding = hor_pad_or_crop
            else:
              self.ver_conv_crop_layer = CropLayer(crop_set=ver_pad_or_crop)
```

```
            ver_conv_padding = (0, 0)
          self.hor_conv_crop_layer = CropLayer(crop_set=hor_pad_or_crop)
            hor_conv_padding = (0, 0)
        self.ver_conv = nn.Conv2d(in_channels=in_channels,
                                  out_channels=out_channels,
                                  kernel_size=(3, 1),
                                  stride=stride,
                                  padding=ver_conv_padding,
                                  dilation=dilation,
                                  groups=groups,
                                  bias=False,
                                  padding_mode=padding_mode)

        self.hor_conv = nn.Conv2d(in_channels=in_channels,
                                  out_channels=out_channels,
                                  kernel_size=(1, 3),
                                  stride=stride,
                                  padding=hor_conv_padding,
                                  dilation=dilation,
                                  groups=groups,
                                  bias=False,
                                  padding_mode=padding_mode)
        self.ver_bn = nn.BatchNorm2d(num_features=out_channels)
        self.hor_bn = nn.BatchNorm2d(num_features=out_channels)

    def forward(self, input):
        if self.deploy:
            return self.fused_conv(input)
        else:
            square_outputs = self.square_conv(input)
            square_outputs = self.square_bn(square_outputs)
            # print(square_outputs.size())
            # return square_outputs
            vertical_outputs = self.ver_conv_crop_layer(input)
            vertical_outputs = self.ver_conv(vertical_outputs)
            vertical_outputs = self.ver_bn(vertical_outputs)
            # print(vertical_outputs.size())
            horizontal_outputs = self.hor_conv_crop_layer(input)
            horizontal_outputs = self.hor_conv(horizontal_outputs)
            horizontal_outputs = self.hor_bn(horizontal_outputs)
            # print(horizontal_outputs.size())
            return square_outputs + vertical_outputs + horizontal_outputs
```

3.2 ASPP



说明：ASPP 是 DeepLabv3+ 其中一个核心创新点，用空间金字塔池化模块来进一步提取多尺度信息，这里是采用不同 rate 的空洞卷积来实现这一点。

论文：https://arxiv.org/pdf/1802.02611

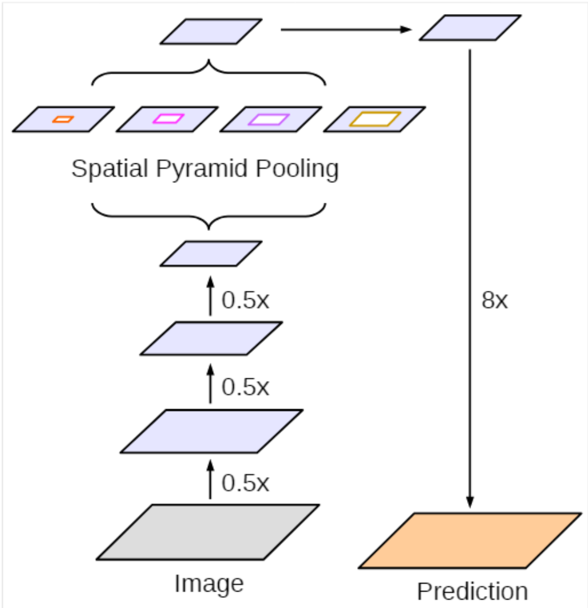
代码：

```
import torch.nn as nn
import torch


class SeparableConv2d(nn.Module):
    def __init__(self,
                 in_channels,
                 out_channels,
                 kernel_size=1,
                 stride=1,
                 padding=0,
                 dilation=1,
                 bias=False):
        super(SeparableConv2d, self).__init__()

        self.conv1 = nn.Conv2d(in_channels,
                               in_channels,
```

```
                                     kernel_size,
                                     stride,
                                     padding,
                                     dilation,
                                     groups=in_channels,
                                     bias=bias)
        self.pointwise = nn.Conv2d(in_channels,
                                         out_channels,
                                         1,
                                         1,
                                         0,
                                         1,
                                         1,
                                         bias=bias)

    def forward(self, x):
        x = self.conv1(x)
        x = self.pointwise(x)
        return x


class ASPP(nn.Module):
    def __init__(self, inplanes, planes, rate):
        super(ASPP, self).__init__()
        self.rate = rate
        if rate == 1:
            kernel_size = 1
            padding = 0
        else:
            kernel_size = 3
            padding = rate
            #self.conv1 = nn.Conv2d(planes, planes, kernel_size=3,
             ↪ bias=False,padding=1)
            self.conv1 = SeparableConv2d(planes, planes, 3, 1, 1)
            self.bn1 = nn.BatchNorm2d(planes)
            self.relu1 = nn.ReLU()

            # self.atrous_convolution = nn.Conv2d(inplanes, planes,
             ↪ kernel_size=kernel_size,
            #                          stride=1, padding=padding,
             ↪ dilation=rate, bias=False)
        self.atrous_convolution = SeparableConv2d(inplanes, planes,
                                                  kernel_size, 1, padding,
```

```
                                                rate)
        self.bn = nn.BatchNorm2d(planes)
        self.relu = nn.ReLU()

        self._init_weight()

    def forward(self, x):
        x = self.atrous_convolution(x)
        x = self.bn(x)
        #x = self.relu(x)
        if self.rate != 1:
            x = self.conv1(x)
            x = self.bn1(x)
            x = self.relu1(x)
        return x

    def _init_weight(self):
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                torch.nn.init.kaiming_normal_(m.weight)
            elif isinstance(m, nn.BatchNorm2d):
                m.weight.data.fill_(1)
                m.bias.data.zero_()
```

3.3 SPP



(a) Spatial Pyramid Pooling

说明：这里 SPP 首先还是在 yolov3-spp 中提出的，借鉴了 SPP-Net 的处理方式，但是实际上有很大差别。

论文：https://github.com/AlexeyAB/darknet

http://pjreddie.com/darknet/

代码（实际就是几个最大池化层进行的组合）：

```
### SPP ###
[maxpool]
stride=1
size=5

[route]
layers=-2

[maxpool]
stride=1
size=9

[route]
layers=-4
```

```
[maxpool]
stride=1
size=13

[route]
layers=-1,-3,-5,-6

### End SPP ###
```

3.4 BlazeBlock

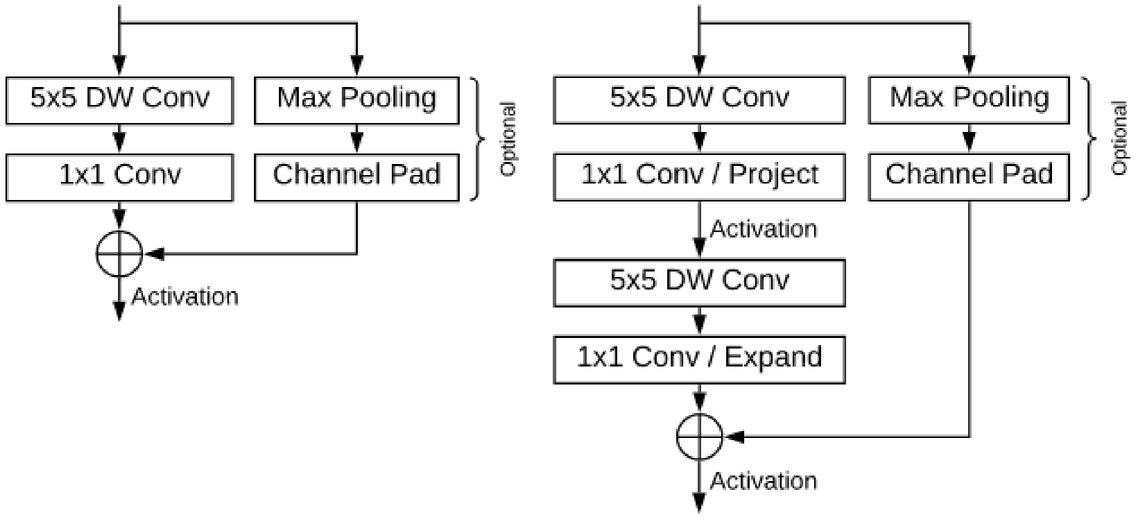


Figure 1. BlazeBlock (left) and double BlazeBlock

说明：来自 BlazeFace 的一个模块，主要作用是轻量化

论文：https://www.arxiv.org/pdf/1907.05047

代码：

```
class BlazeBlock(nn.Module):
    def __init__(self, inp, oup1, oup2=None, stride=1, kernel_size=5):
        super(BlazeBlock, self).__init__()
        self.stride = stride
        assert stride in [1, 2]

        self.use_double_block = oup2 is not None
```

```
        self.use_pooling = self.stride != 1

        if self.use_double_block:
            self.channel_pad = oup2 - inp
        else:
            self.channel_pad = oup1 - inp

        padding = (kernel_size - 1) // 2

        self.conv1 = nn.Sequential(
            # dw
            nn.Conv2d(inp, inp, kernel_size=kernel_size, stride=stride,
                      padding=padding, groups=inp, bias=True),
            nn.BatchNorm2d(inp),
            # pw-linear
            nn.Conv2d(inp, oup1, 1, 1, 0, bias=True),
            nn.BatchNorm2d(oup1),
        )
        self.act = nn.ReLU(inplace=True)

        if self.use_double_block:
            self.conv2 = nn.Sequential(
                nn.ReLU(inplace=True),
                # dw
                nn.Conv2d(oup1, oup1, kernel_size=kernel_size,
                          stride=1, padding=padding, groups=oup1, bias=True),
                nn.BatchNorm2d(oup1),
                # pw-linear
                nn.Conv2d(oup1, oup2, 1, 1, 0, bias=True),
                nn.BatchNorm2d(oup2),
            )

        if self.use_pooling:
            self.mp = nn.MaxPool2d(kernel_size=self.stride,
             ↪  stride=self.stride)

    def forward(self, x):
        h = self.conv1(x)
        if self.use_double_block:
            h = self.conv2(h)

        # skip connection
        if self.use_pooling:
```

```
            x = self.mp(x)
        if self.channel_pad > 0:
            x = F.pad(x, (0, 0, 0, 0, 0, self.channel_pad), 'constant', 0)
        return self.act(h + x)


def initialize(module):
    # original implementation is unknown
    if isinstance(module, nn.Conv2d):
        nn.init.kaiming_normal_(module.weight.data)
        nn.init.constant_(module.bias.data, 0)
    elif isinstance(module, nn.BatchNorm2d):
        nn.init.constant_(module.weight.data, 1)
        nn.init.constant_(module.bias.data, 0)
```

3.5 深度可分离卷积

这个都比较熟悉，直接上代码：

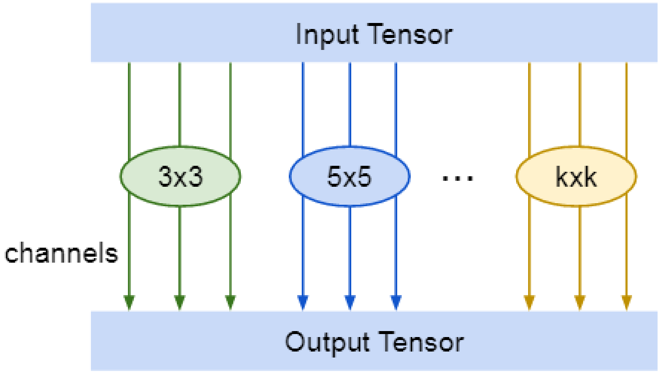
```
import torch.nn as nn


class DWConv(nn.Module):
    def __init__(self, in_plane, out_plane):
        super(DWConv, self).__init__()
        self.depth_conv = nn.Conv2d(in_channels=in_plane,
                                    out_channels=in_plane,
                                    kernel_size=3,
                                    stride=1,
                                    padding=1,
                                    groups=in_plane)
        self.point_conv = nn.Conv2d(in_channels=in_plane,
                                    out_channels=out_plane,
                                    kernel_size=1,
                                    stride=1,
                                    padding=0,
                                    groups=1)
    def forward(self, x):
        x = self.depth_conv(x)
        x = self.point_conv(x)
        return x
```

3.6 FuseConvBn

折叠 BN 在公众号历史文章中可以看详解，用于在推理过程中加速推理过程。

```
import torch

def fuse_conv_and_bn(conv, bn):
    # https://tehnokv.com/posts/fusing-batchnorm-and-conv/
    with torch.no_grad():
        # init
        fusedconv = torch.nn.Conv2d(conv.in_channels,
                                    conv.out_channels,
                                    kernel_size=conv.kernel_size,
                                    stride=conv.stride,
                                    padding=conv.padding,
                                    bias=True)

        # prepare filters
        w_conv = conv.weight.clone().view(conv.out_channels, -1)
       w_bn = torch.diag(bn.weight.div(torch.sqrt(bn.eps + bn.running_var)))
        fusedconv.weight.copy_(torch.mm(w_bn,
↪   w_conv).view(fusedconv.weight.size()))

        # prepare spatial bias
        if conv.bias is not None:
            b_conv = conv.bias
        else:
            b_conv = torch.zeros(conv.weight.size(0))
        b_bn = bn.bias -
↪   bn.weight.mul(bn.running_mean).div(torch.sqrt(bn.running_var + bn.eps))
        fusedconv.bias.copy_(torch.mm(w_bn, b_conv.reshape(-1,
↪   1)).reshape(-1) + b_bn)

        return fusedconv
```

3.7 MixConv2d



(b) Our proposed MixConv

说明：这个模块是在 MixNet 中提出的，使用 AutoML 搜索的情况下，对卷积核进行了搜索和调整。

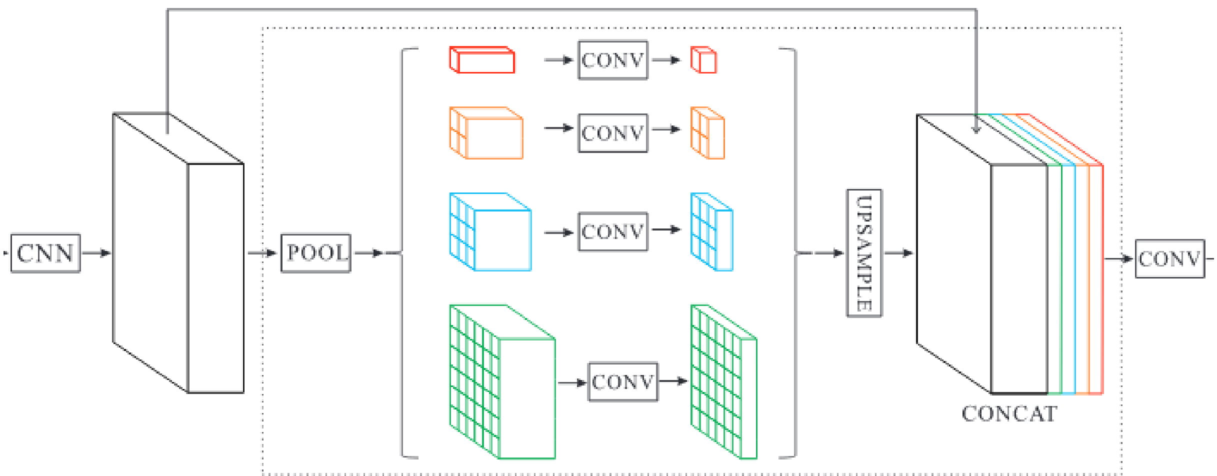
论文：https://arxiv.org/pdf/1907.09595.pdf

代码（以下代码出自 u 版 yolov3）：

```
import numpy as np
import torch
import torch.nn as nn


class MixConv2d(nn.Module):  # MixConv: Mixed Depthwise Convolutional Kernels https://arxiv.org/abs/1907.09595
    def __init__(self, in_ch, out_ch, k=(3, 5, 7), stride=1, dilation=1, bias=True, method='equal_params'):
        super(MixConv2d, self).__init__()

        groups = len(k)
        if method == 'equal_ch':  # equal channels per group
            i = torch.linspace(0, groups - 1E-6, out_ch).floor()  # out_ch indices
            ch = [(i == g).sum() for g in range(groups)]
        else:  # 'equal_params': equal parameter count per group
            b = [out_ch] + [0] * groups
            a = np.eye(groups + 1, groups, k=-1)
            a -= np.roll(a, 1, axis=1)
            a *= np.array(k) ** 2
            a[0] = 1
```

```
            ch = np.linalg.lstsq(a, b, rcond=None)[0].round().astype(int)  #
↪   solve for equal weight indices, ax = b

        self.m = nn.ModuleList([nn.Conv2d(in_channels=in_ch,
                                          out_channels=ch[g],
                                          kernel_size=k[g],
                                          stride=stride,
                                          padding=k[g] // 2,  # 'same' pad
                                          dilation=dilation,
                                         bias=bias) for g in range(groups)])

    def forward(self, x):
        return torch.cat([m(x) for m in self.m], 1)
```

3.8 PPM



(b) Feature Map (c) Pyramid Pooling Module

说明：跟 ASPP 类似，只不过 PSPNet 的 PPM 是使用了池化进行的融合特征金字塔，聚合不同区域的上下文信息。

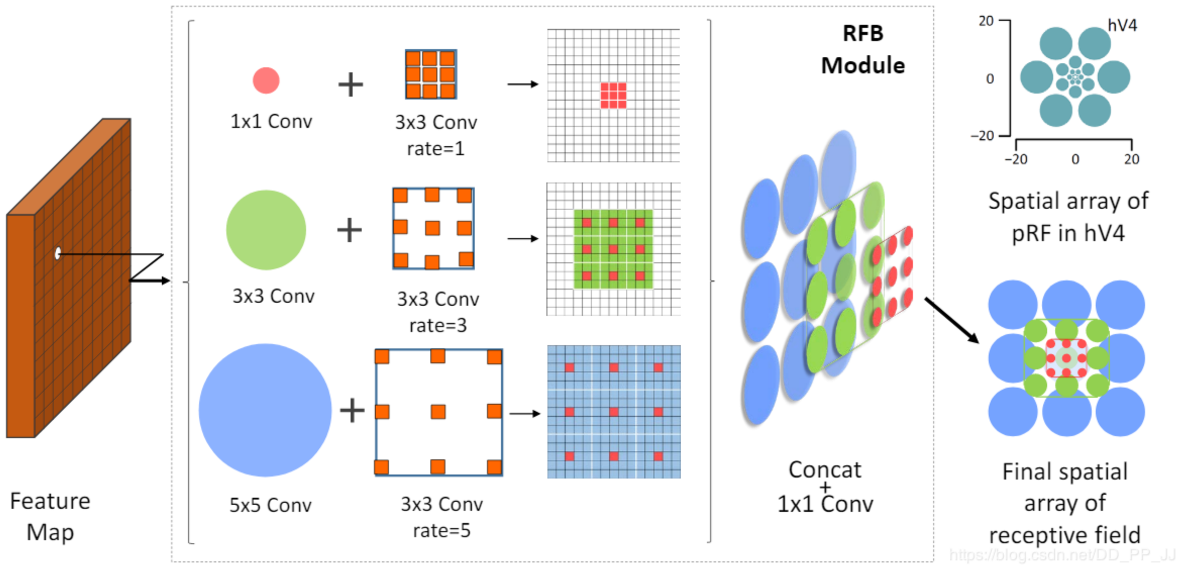
论文：https://arxiv.org/abs/1612.01105

代码：

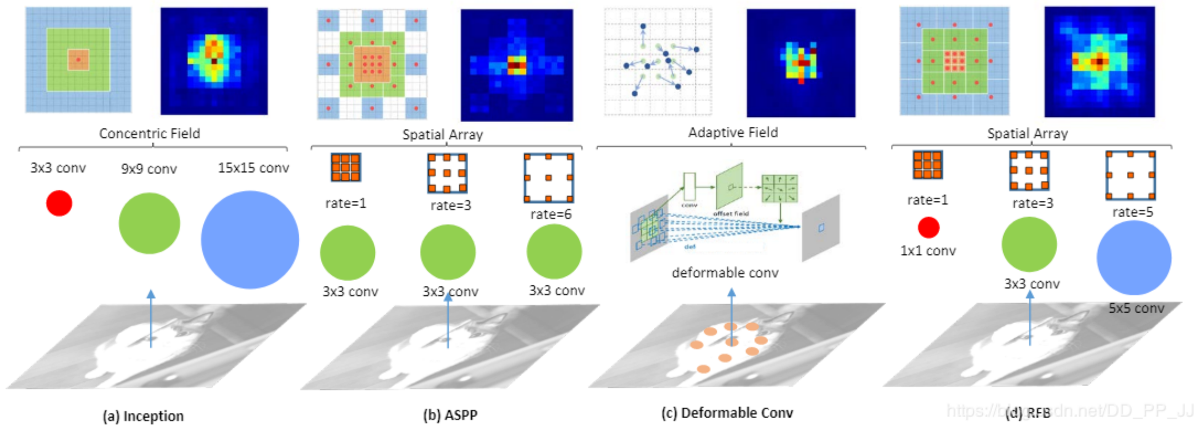
```
import torch.nn as nn
import torch
import torch.nn.functional as F

class PSPModule(nn.Module):
```

```python
def __init__(self, features, out_features=1024, sizes=(1, 2, 3, 6)):
    super().__init__()
    self.stages = []
    self.stages = nn.ModuleList(
        [self._make_stage(features, size) for size in sizes])
    self.bottleneck = nn.Conv2d(features * (len(sizes) + 1),
                                out_features,
                                kernel_size=1)
    self.relu = nn.ReLU()

def _make_stage(self, features, size):
    prior = nn.AdaptiveAvgPool2d(output_size=(size, size))
    conv = nn.Conv2d(features, features, kernel_size=1, bias=False)
    return nn.Sequential(prior, conv)

def forward(self, feats):
    h, w = feats.size(2), feats.size(3)
    priors = [
        F.upsample(input=stage(feats), size=(h, w), mode='bilinear')
        for stage in self.stages
    ] + [feats]
    bottle = self.bottleneck(torch.cat(priors, 1))
    return self.relu(bottle)
```

3.9 RFB



说明：RFBNet 提出了两种 RFB 模型，RFB 和 RFB-s, 分别用于深层和浅层。和 ASPP，PPM 类似。来看一个对比图：



论文：https://arxiv.org/abs/1711.07767

代码：

```
import torch.nn as nn
import torch


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(inplace=True) 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

class BasicRFB(nn.Module):
    '''
    [rfb]
    filters = 128
    stride = 1 or 2
    scale = 1.0
    '''
    def __init__(self, in_planes, out_planes, stride=1, scale=0.1, visual=1):
        super(BasicRFB, self).__init__()
        self.scale = scale
        self.out_channels = out_planes
        inter_planes = in_planes // 8
        self.branch0 = nn.Sequential(
            BasicConv(in_planes,
                      2 * inter_planes,
                      kernel_size=1,
                      stride=stride),
            BasicConv(2 * inter_planes,
                      2 * inter_planes,
                      kernel_size=3,
                      stride=1,
                      padding=visual,
                      dilation=visual,
                      relu=False))
        self.branch1 = nn.Sequential(
            BasicConv(in_planes, inter_planes, kernel_size=1, stride=1),
            BasicConv(inter_planes,
                      2 * inter_planes,
                      kernel_size=(3, 3),
                      stride=stride,
```

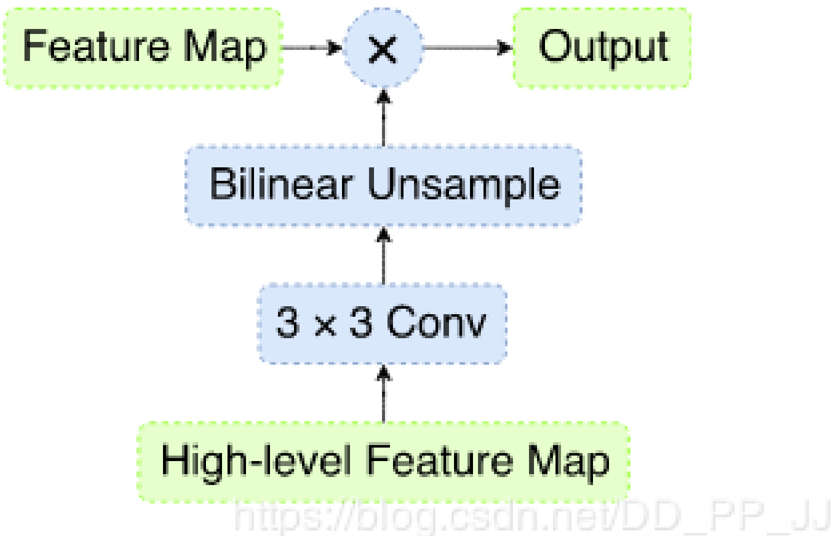
```
                          padding=(1, 1)),
                BasicConv(2 * inter_planes,
                          2 * inter_planes,
                          kernel_size=3,
                          stride=1,
                          padding=visual + 1,
                          dilation=visual + 1,
                          relu=False))
        self.branch2 = nn.Sequential(
            BasicConv(in_planes, inter_planes, kernel_size=1, stride=1),
            BasicConv(inter_planes, (inter_planes // 2) * 3,
                          kernel_size=3,
                          stride=1,
                          padding=1),
            BasicConv((inter_planes // 2) * 3,
                          2 * inter_planes,
                          kernel_size=3,
                          stride=stride,
                          padding=1),
            BasicConv(2 * inter_planes,
                          2 * inter_planes,
                          kernel_size=3,
                          stride=1,
                          padding=2 * visual + 1,
                          dilation=2 * visual + 1,
                          relu=False))

        self.ConvLinear = BasicConv(6 * inter_planes,
                                    out_planes,
                                    kernel_size=1,
                                    stride=1,
                                    relu=False)
        self.shortcut = BasicConv(in_planes,
                                  out_planes,
                                  kernel_size=1,
                                  stride=stride,
                                  relu=False)
        self.relu = nn.ReLU(inplace=False)

    def forward(self, x):
        x0 = self.branch0(x)
        x1 = self.branch1(x)
        x2 = self.branch2(x)
```

```
        out = torch.cat((x0, x1, x2), 1)
        out = self.ConvLinear(out)
        short = self.shortcut(x)
        out = out * self.scale + short
        out = self.relu(out)

        return out


class BasicRFB_small(nn.Module):
    '''
    [rfbs]
    filters = 128
    stride=1 or 2
    scale = 1.0
    '''
    def __init__(self, in_planes, out_planes, stride=1, scale=0.1):
        super(BasicRFB_small, self).__init__()
        self.scale = scale
        self.out_channels = out_planes
        inter_planes = in_planes // 4

        self.branch0 = nn.Sequential(
            BasicConv(in_planes, inter_planes, kernel_size=1, stride=1),
            BasicConv(inter_planes,
                      inter_planes,
                      kernel_size=3,
                      stride=1,
                      padding=1,
                      relu=False))
        self.branch1 = nn.Sequential(
            BasicConv(in_planes, inter_planes, kernel_size=1, stride=1),
            BasicConv(inter_planes,
                      inter_planes,
                      kernel_size=(3, 1),
                      stride=1,
                      padding=(1, 0)),
            BasicConv(inter_planes,
                      inter_planes,
                      kernel_size=3,
                      stride=1,
                      padding=3,
```

```
                          dilation=3,
                          relu=False))
        self.branch2 = nn.Sequential(
            BasicConv(in_planes, inter_planes, kernel_size=1, stride=1),
            BasicConv(inter_planes,
                      inter_planes,
                      kernel_size=(1, 3),
                      stride=stride,
                      padding=(0, 1)),
            BasicConv(inter_planes,
                      inter_planes,
                      kernel_size=3,
                      stride=1,
                      padding=3,
                      dilation=3,
                      relu=False))
        self.branch3 = nn.Sequential(
           BasicConv(in_planes, inter_planes // 2, kernel_size=1, stride=1),
            BasicConv(inter_planes // 2, (inter_planes // 4) * 3,
                      kernel_size=(1, 3),
                      stride=1,
                      padding=(0, 1)),
            BasicConv((inter_planes // 4) * 3,
                      inter_planes,
                      kernel_size=(3, 1),
                      stride=stride,
                      padding=(1, 0)),
            BasicConv(inter_planes,
                      inter_planes,
                      kernel_size=3,
                      stride=1,
                      padding=5,
                      dilation=5,
                      relu=False))

        self.ConvLinear = BasicConv(4 * inter_planes,
                                    out_planes,
                                    kernel_size=1,
                                    stride=1,
                                    relu=False)
        self.shortcut = BasicConv(in_planes,
                                  out_planes,
                                  kernel_size=1,
```

```
                                  stride=stride,
                                  relu=False)
        self.relu = nn.ReLU(inplace=False)

    def forward(self, x):
        x0 = self.branch0(x)
        x1 = self.branch1(x)
        x2 = self.branch2(x)
        x3 = self.branch3(x)

        out = torch.cat((x0, x1, x2, x3), 1)
        out = self.ConvLinear(out)
        short = self.shortcut(x)
        out = out * self.scale + short
        out = self.relu(out)

        return out
```

3.10 SEB



说明：严格来说，这不属于即插即用模块，但是我比较喜欢这种简单而实用的构造，所以也加进来了。SEB 是 ExFuse 论文中提出的一种特征融合方法，并没有采用传统的相加或者 concatenation 的方法，使用了相乘的方法。

论文：https://arxiv.org/pdf/1804.03821

代码：

```
class SematicEmbbedBlock(nn.Module):
    def __init__(self, high_in_plane, low_in_plane, out_plane):
        super(SematicEmbbedBlock, self).__init__()
        self.conv3x3 = nn.Conv2d(high_in_plane, out_plane, 3, 1, 1)
        self.upsample = nn.UpsamplingBilinear2d(scale_factor=2)

        self.conv1x1 = nn.Conv2d(low_in_plane, out_plane, 1)

    def forward(self, high_x, low_x):
        high_x = self.upsample(self.conv3x3(high_x))
        low_x = self.conv1x1(low_x)
        return high_x * low_x
```

3.11 SSHContextModule

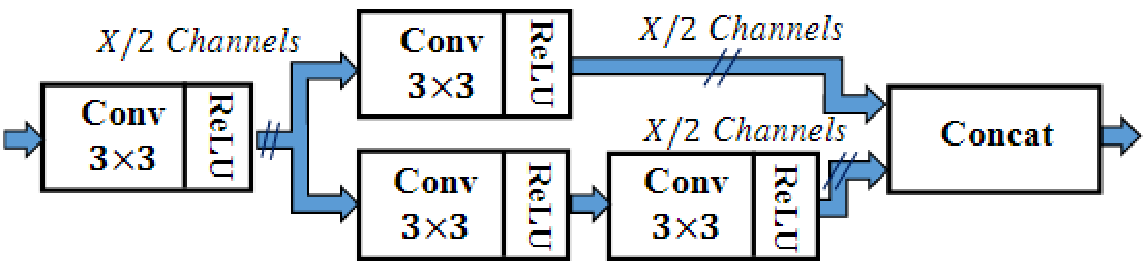


Figure 4: *SSH* context module.

说明：其实这样一看这个模块就是利用了两个分支不同的感受野，然后进行了融合，使用在人脸识别中，一个小的模块。

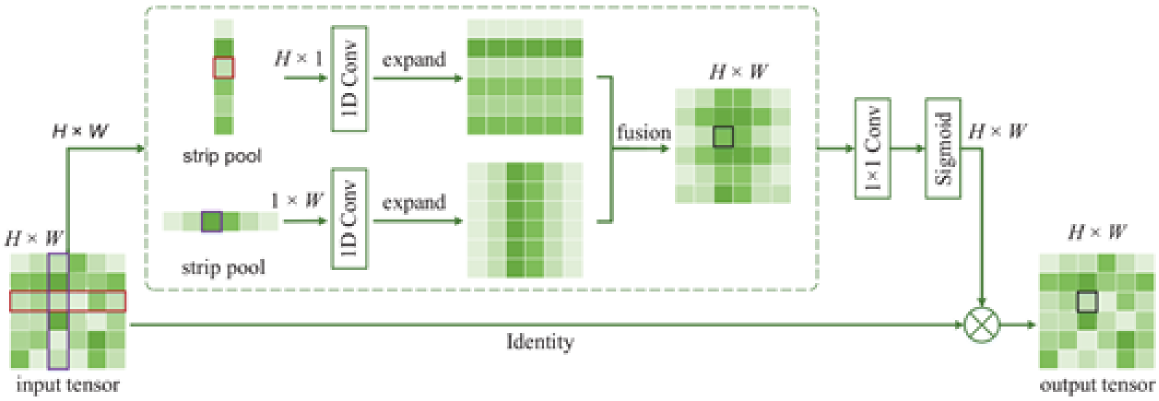
论文：https://www.arxiv.org/pdf/1708.03979

论文：

```
import torch
import torch.nn as nn

class Conv3x3BNReLU(nn.Module):
    def __init__(self, in_channel, out_channel):
        super(Conv3x3BNReLU,self).__init__()
        self.conv3x3 = nn.Conv2d(in_channel, out_channel, 3, 1, 1)
```

```python
        self.bn = nn.BatchNorm2d(out_channel)
        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
        return self.relu(self.bn(self.conv3x3(x)))


class SSHContextModule(nn.Module):
    def __init__(self, in_channel):
        super(SSHContextModule, self).__init__()
        self.stem = Conv3x3BNReLU(in_channel, in_channel//2)
        self.branch1_conv3x3 = Conv3x3BNReLU(in_channel//2, in_channel//2)
        self.branch2_conv3x3_1 = Conv3x3BNReLU(in_channel//2, in_channel//2)
        self.branch2_conv3x3_2 = Conv3x3BNReLU(in_channel//2, in_channel//2)

    def forward(self, x):
        x = self.stem(x)
        # branch1
        x1 = self.branch1_conv3x3(x)
        # branch2
        x2 = self.branch2_conv3x3_1(x)
        x2 = self.branch2_conv3x3_2(x2)
        # concat
        # print(x1.shape, x2.shape)
        return torch.cat([x1, x2], dim=1)

if __name__ == "__main__":
    in_tensor = torch.zeros((6, 64, 128, 128))
    module = SSHContextModule(64)
    out_tensor = module(in_tensor)
    print(out_tensor.shape)
```

3.12 Strip Pooling



说明：跟 CCNet 挺像的，就是对 SPP 这种传统的 Spatial Pooling 进行了改进，设计了新的体系结构。

论文：https://arxiv.org/abs/2003.13328v1

代码：

```
import torch
import torch.nn as nn

import torch.nn.functional as F

'''
https://www.cnblogs.com/YongQiVisionIMAX/p/12630769.html
https://github.com/Andrew-Qibin/SPNet/blob/master/models/spnet.py
'''


class StripPooling(nn.Module):
    def __init__(self, in_channels, pool_size, norm_layer, up_kwargs):
        super(StripPooling, self).__init__()
        self.pool1 = nn.AdaptiveAvgPool2d(pool_size[0])
        self.pool2 = nn.AdaptiveAvgPool2d(pool_size[1])

        self.pool3 = nn.AdaptiveAvgPool2d((1, None))
        self.pool4 = nn.AdaptiveAvgPool2d((None, 1))

        inter_channels = int(in_channels/4)
```

```
        self.conv1_1 = nn.Sequential(nn.Conv2d(in_channels, inter_channels,
         ↪ 1, bias=False),
                                     norm_layer(inter_channels),
                                     nn.ReLU(True))
        self.conv1_2 = nn.Sequential(nn.Conv2d(in_channels, inter_channels,
         ↪ 1, bias=False),
                                     norm_layer(inter_channels),
                                     nn.ReLU(True))

        self.conv2_0 = nn.Sequential(nn.Conv2d(inter_channels,
         ↪ inter_channels, 3, 1, 1, bias=False),
                                     norm_layer(inter_channels))
        self.conv2_1 = nn.Sequential(nn.Conv2d(inter_channels,
         ↪ inter_channels, 3, 1, 1, bias=False),
                                     norm_layer(inter_channels))
        self.conv2_2 = nn.Sequential(nn.Conv2d(inter_channels,
         ↪ inter_channels, 3, 1, 1, bias=False),
                                     norm_layer(inter_channels))
        self.conv2_3 = nn.Sequential(nn.Conv2d(inter_channels,
         ↪ inter_channels, (1, 3), 1, (0, 1), bias=False),
                                     norm_layer(inter_channels))
        self.conv2_4 = nn.Sequential(nn.Conv2d(inter_channels,
         ↪ inter_channels, (3, 1), 1, (1, 0), bias=False),
                                     norm_layer(inter_channels))
        self.conv2_5 = nn.Sequential(nn.Conv2d(inter_channels,
         ↪ inter_channels, 3, 1, 1, bias=False),
                                     norm_layer(inter_channels),
                                     nn.ReLU(True))
        self.conv2_6 = nn.Sequential(nn.Conv2d(inter_channels,
         ↪ inter_channels, 3, 1, 1, bias=False),
                                     norm_layer(inter_channels),
                                     nn.ReLU(True))
        self.conv3 = nn.Sequential(nn.Conv2d(inter_channels*2, in_channels,
         ↪ 1, bias=False),
                                   norm_layer(in_channels))
        # bilinear interpolate options
        self._up_kwargs = up_kwargs

    def forward(self, x):
        _, _, h, w = x.size()
        x1 = self.conv1_1(x)
        x2 = self.conv1_2(x)
```

```
        x2_1 = self.conv2_0(x1)

        x2_2 = F.interpolate(self.conv2_1(self.pool1(x1)),
                             (h, w), **self._up_kwargs)
        x2_3 = F.interpolate(self.conv2_2(self.pool2(x1)),
                             (h, w), **self._up_kwargs)
        x2_4 = F.interpolate(self.conv2_3(self.pool3(x2)),
                             (h, w), **self._up_kwargs)
        x2_5 = F.interpolate(self.conv2_4(self.pool4(x2)),
                             (h, w), **self._up_kwargs)

        x1 = self.conv2_5(F.relu_(x2_1 + x2_2 + x2_3))
        x2 = self.conv2_6(F.relu_(x2_5 + x2_4))
        out = self.conv3(torch.cat([x1, x2], dim=1))

        return F.relu_(x + out)


class PyramidPooling(nn.Module):
    """
    Reference:
        Zhao, Hengshuang, et al. *"Pyramid scene parsing network."*
    """

    def __init__(self, in_channels, norm_layer, up_kwargs):
        super(PyramidPooling, self).__init__()
        self.pool1 = nn.AdaptiveAvgPool2d(1)
        self.pool2 = nn.AdaptiveAvgPool2d(2)
        self.pool3 = nn.AdaptiveAvgPool2d(3)
        self.pool4 = nn.AdaptiveAvgPool2d(6)

        out_channels = int(in_channels/4)
        self.conv1 = nn.Sequential(nn.Conv2d(in_channels, out_channels, 1,
         ↪ bias=False),
                                   norm_layer(out_channels),
                                   nn.ReLU(True))
        self.conv2 = nn.Sequential(nn.Conv2d(in_channels, out_channels, 1,
         ↪ bias=False),
                                   norm_layer(out_channels),
                                   nn.ReLU(True))
        self.conv3 = nn.Sequential(nn.Conv2d(in_channels, out_channels, 1,
         ↪ bias=False),
                                   norm_layer(out_channels),
```

```
                                    nn.ReLU(True))
        self.conv4 = nn.Sequential(nn.Conv2d(in_channels, out_channels, 1,
         ↪  bias=False),
                                    norm_layer(out_channels),
                                    nn.ReLU(True))
        # bilinear interpolate options
        self._up_kwargs = up_kwargs

    def forward(self, x):
        _, _, h, w = x.size()
        feat1 = F.interpolate(self.conv1(self.pool1(x)),
                              (h, w), **self._up_kwargs)
        feat2 = F.interpolate(self.conv2(self.pool2(x)),
                              (h, w), **self._up_kwargs)
        feat3 = F.interpolate(self.conv3(self.pool3(x)),
                              (h, w), **self._up_kwargs)
        feat4 = F.interpolate(self.conv4(self.pool4(x)),
                              (h, w), **self._up_kwargs)
        return torch.cat((x, feat1, feat2, feat3, feat4), 1)


class SPHead(nn.Module):
    def __init__(self, in_channels, out_channels, norm_layer, up_kwargs):
        super(SPHead, self).__init__()
        inter_channels = in_channels // 2
        self.trans_layer = nn.Sequential(nn.Conv2d(in_channels,
         ↪  inter_channels, 1, 1, 0, bias=False),
                                         norm_layer(inter_channels),
                                         nn.ReLU(True)
                                         )
        self.strip_pool1 = StripPooling(
            inter_channels, (20, 12), norm_layer, up_kwargs)
        self.strip_pool2 = StripPooling(
            inter_channels, (20, 12), norm_layer, up_kwargs)
        self.score_layer = nn.Sequential(nn.Conv2d(inter_channels,
         ↪  inter_channels // 2, 3, 1, 1, bias=False),
                                         norm_layer(inter_channels // 2),
                                         nn.ReLU(True),
                                         nn.Dropout2d(0.1, False),
                                         nn.Conv2d(inter_channels // 2,
 ↪  out_channels, 1))

    def forward(self, x):
```

```
        x = self.trans_layer(x)
        x = self.strip_pool1(x)
        x = self.strip_pool2(x)
        x = self.score_layer(x)
        return x
```