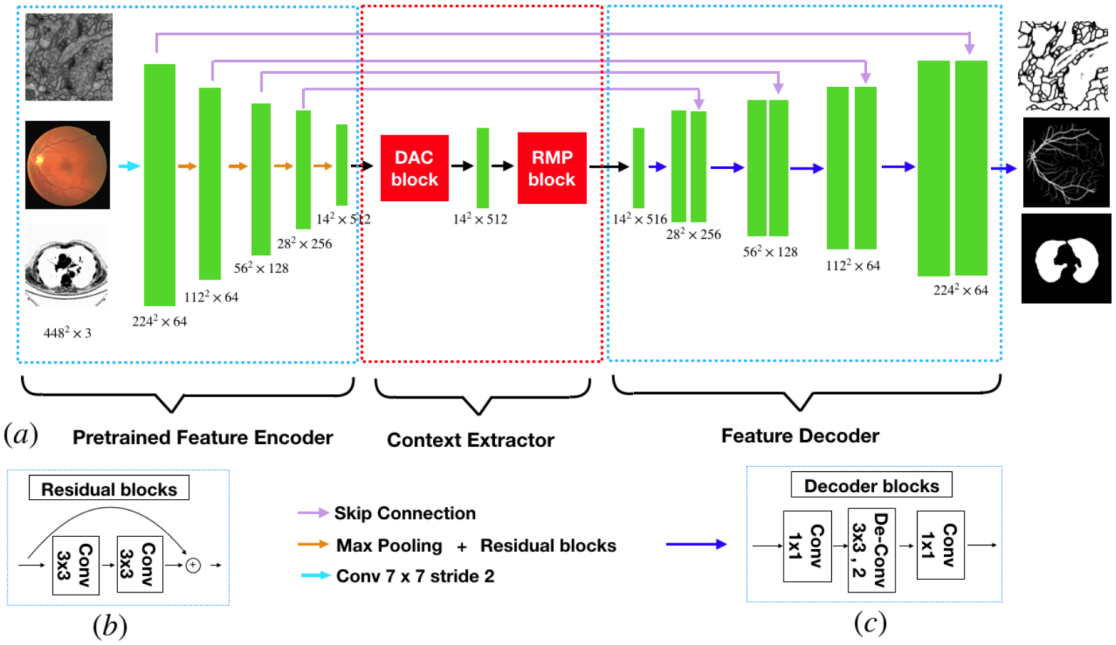
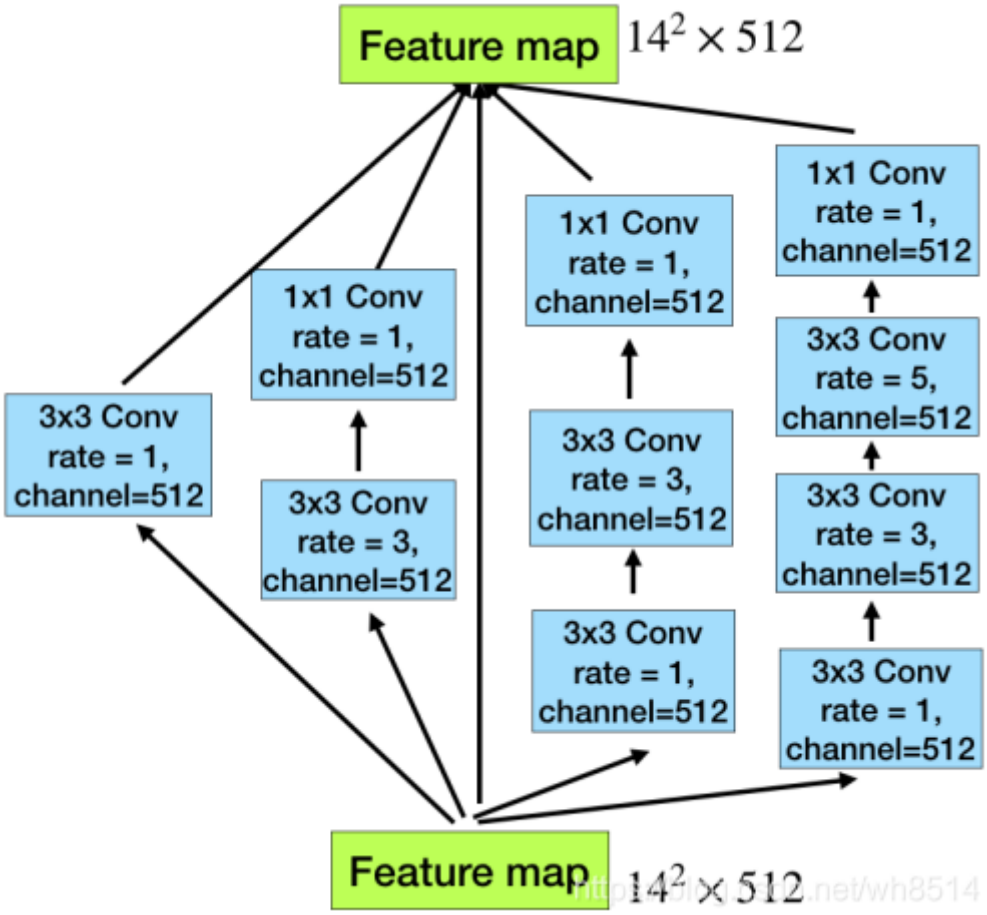
论文

CE-Net



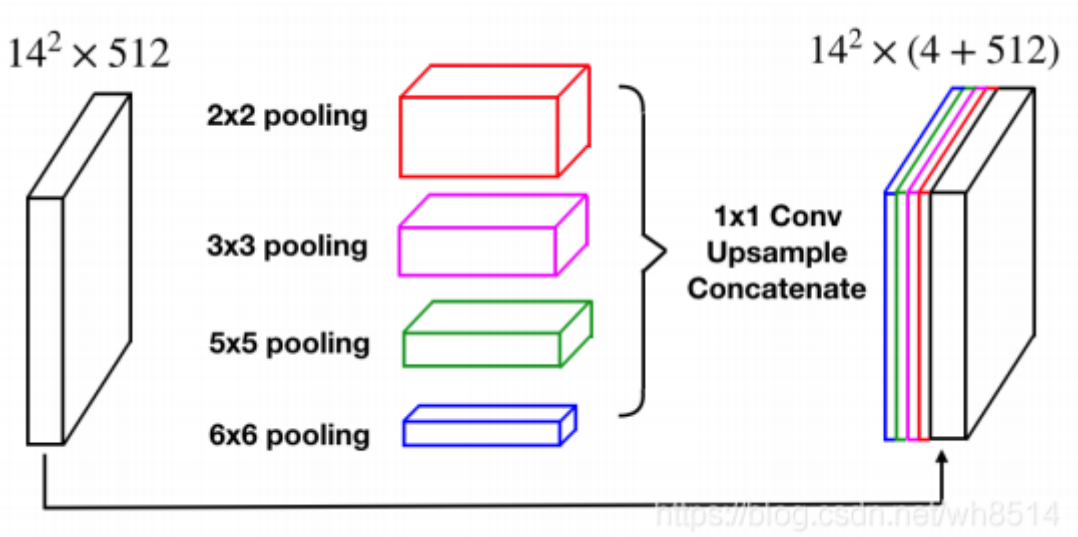
U-Net及其变体的一个常见问题在于连续的池化和跨度卷积降低了特征分辨率，以学习越来越抽象的特征表达。虽然，这种不变性有利于分类或物体检测任务，但是它常常不利于需要详细空间信息的密集预测任务。U-Net可以被看作编码器-解码器结构，编码器旨在逐步减少feature map的空间维度和捕获更多高级语义特征；解码器旨在恢复目标细节信息和空间维度。CE-Net则可以填补这个缺点，相较于Unet可以捕获更多高级信息并保留空间信息用于2D医学图像分割，主要包含三个模块：

- 特征编码层
  作者使用的是预训练的**ResNet**作为固定特征提取器，在U-Net架构中，每个编码器块包含两个卷积层和一个最大池化层。在论文中，作者将其替换为预训练好的ResNet-34在特征编码器模块中，该模块保留了前四个特征提取块，而**没有均值池化和全连接**。与原始块相比，ResNet**增加了快捷机制**，从而避免了梯度消失问题，并加速了网络收敛。
- 上下文提取模块
  上下文提取模块由**DAC**(密集的空洞卷积块)模块和**RMP**（残差多核池化块）模块组成，作者提出了密集空洞卷积块(DAC)来应用空洞卷积，原始的U-Net架构通过在编码过程中采用连续的3*3卷积和池化操作来捕获多尺度特征，而作者提出的DAC模块可以通过注入具有多尺度空洞卷积的四个**级联分支**来捕获更广泛和更深的语义特征，通常情况下，大接受场的卷积可以为大对象提取和生成更抽象的特征，而小接受场的卷积对于小对象更好。通过合并不同rate的空洞卷积，模块能够对不同大小的对象特征进行提取，得到丰富的特征，在该模块中，使用快捷连接来防止梯度消失问题。




```
class DACblock(nn.Module):
    def __init__(self, channel):
        super(DACblock, self).__init__()
        #三个空洞卷积层 K=K+(K-1)(rate-1)
        self.dilate1 = nn.Conv2d(channel, channel, kernel_size=3, dilation=1,
padding=1)
        self.dilate2 = nn.Conv2d(channel, channel, kernel_size=3, dilation=3,
padding=3)
        self.dilate3 = nn.Conv2d(channel, channel, kernel_size=3, dilation=5,
padding=5)
        #1x1卷积层
        self.conv1x1 = nn.Conv2d(channel, channel, kernel_size=1, dilation=1,
padding=0)
        #modules()是当前模型所有模块的迭代器
        for m in self.modules():
            if isinstance(m, nn.Conv2d) or isinstance(m, nn.ConvTranspose2d):
                if m.bias is not None:
                    m.bias.data.zero_()

    def forward(self, x):
        dilate1_out = nonlinearity(self.dilate1(x))
        dilate2_out = nonlinearity(self.conv1x1(self.dilate2(x)))
        dilate3_out = nonlinearity(self.conv1x1(self.dilate2(self.dilate1(x))))
        dilate4_out =
nonlinearity(self.conv1x1(self.dilate3(self.dilate2(self.dilate1(x)))))
        out = x + dilate1_out + dilate2_out + dilate3_out + dilate4_out
        return out
```

受空间金字塔池化的启发，作者还提出了残差多核池化(RMP)模块。这个RMP模块通过采用各种大小的池化操作进一步编码从DAC中提取对象的多尺度上下文特征，而没有额外的学习权重，依靠多个有效的视场来检测不同大小的对象，以解决分割医学图像物体差异较大的问题。作者提出的**DAC模块使用多尺度的空洞卷积来提取丰富的特征表达，然后RMP模块使用多尺度池化操作来进一步获取上下文信息。**CE-Net的新型上下文编码器网络，它依赖于DAC模块和RMP模块来捕获更多抽象特征和保留更多空间信息以提高医学图像分割的性能。



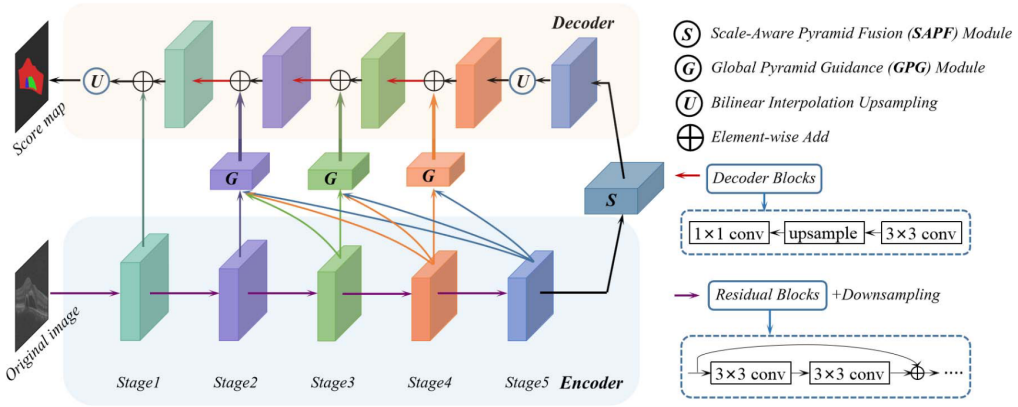

```
class RMPblock(nn.Module):
    def __init__(self, in_channels):
        super(SPPblock, self).__init__()
        self.pool1 = nn.MaxPool2d(kernel_size=[2, 2], stride=2)
        self.pool2 = nn.MaxPool2d(kernel_size=[3, 3], stride=3)
        self.pool3 = nn.MaxPool2d(kernel_size=[5, 5], stride=5)
        self.pool4 = nn.MaxPool2d(kernel_size=[6, 6], stride=6)

        self.conv = nn.Conv2d(in_channels=in_channels, out_channels=1,
kernel_size=1, padding=0)

    def forward(self, x):
        self.in_channels, h, w = x.size(1), x.size(2), x.size(3)
        self.layer1 = F.interpolate(self.conv(self.pool1(x)), size=(h, w),
mode='bilinear', align_corners=True)
        self.layer2 = F.interpolate(self.conv(self.pool2(x)), size=(h, w),
mode='bilinear', align_corners=True)
        self.layer3 = F.interpolate(self.conv(self.pool3(x)), size=(h, w),
mode='bilinear', align_corners=True)
        self.layer4 = F.interpolate(self.conv(self.pool4(x)), size=(h, w),
mode='bilinear', align_corners=True)

        out = torch.cat([self.layer1, self.layer2, self.layer3, self.layer4, x],
1)

        return out
```

- 特征解码模块

快捷连接从编码器到解码器来获取一些详细信息，来补救由于连续池化和卷积操作而导致的信息丢失，简单的上采样和反卷积（转置卷积），反卷积采用卷积操作来放大图像，转置卷积可以学习自适应映射来恢复具有详细信息的特征，作者采用转置卷积来恢复解码器的更高分辨率特征，它主要包括1×1卷积，3×3转置卷积和1×1卷积。

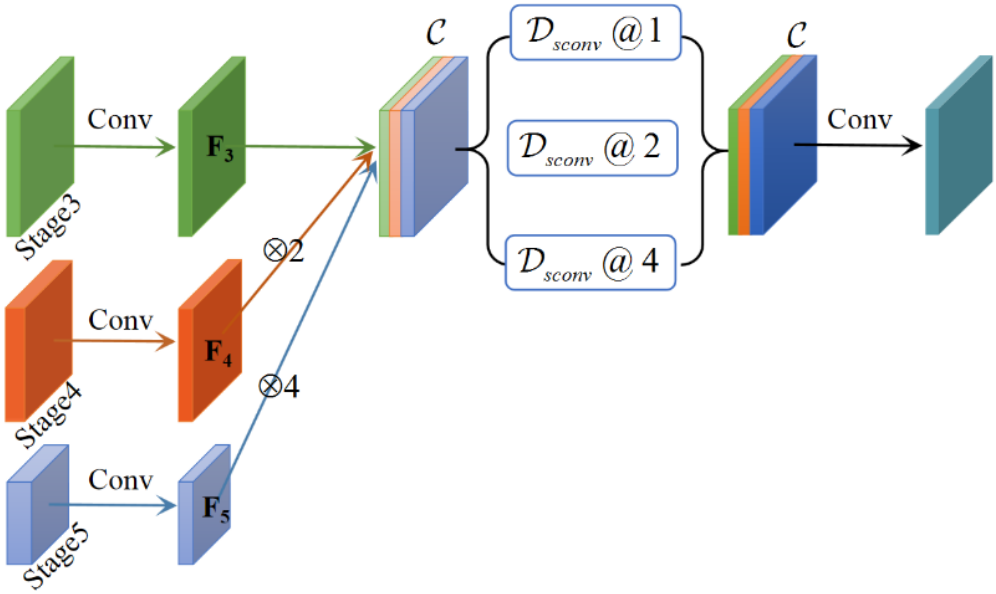
CPFNet

是用在医学影像上的网络，增加了两个金字塔形模型模块来融合全局上下文信息，分别是SAPF模块和GPG模块。



主干还是选用ResNet34,通过SAPF模块捕获并动态合并多尺度信息；跳跃连接层引用GPG模块；解码器采用3*3卷积+双线性上采样+1x1卷积。

- GPG（Global Pyramid Guidance Module）

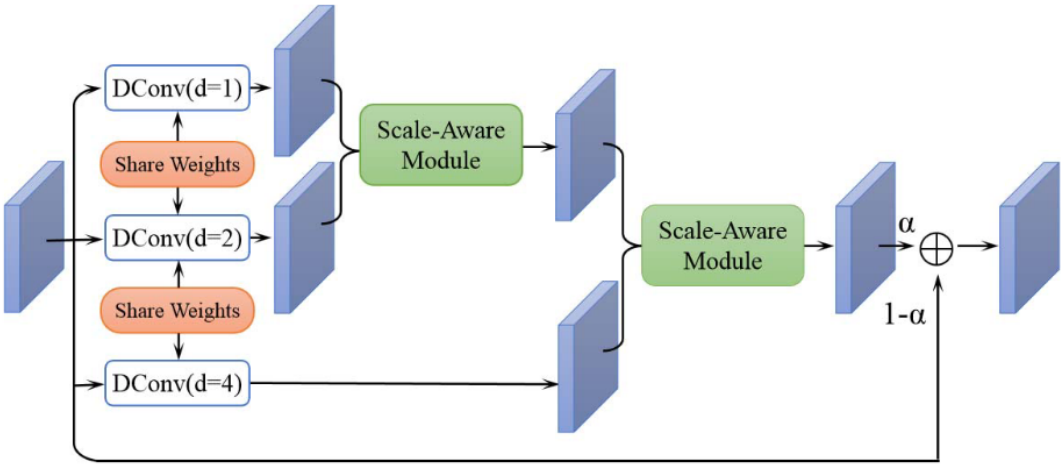


在跳跃连接时，**融合更深层的信息（更高层的语义信息），在解码器中弥补信息的缺少。**先利用卷积层使各层channel数相同，再通过上采样得到相同的尺寸之后拼接起来，再经过空间可分离卷积（可分离卷积用来减少参数），从不同级别的特征映射中提取全局上下文信息，并行使用了三个不同扩张率(1，2和4)的可分离卷积，获取不同感受野的各层融合信息，最后通过3x3卷积得到最终结果。

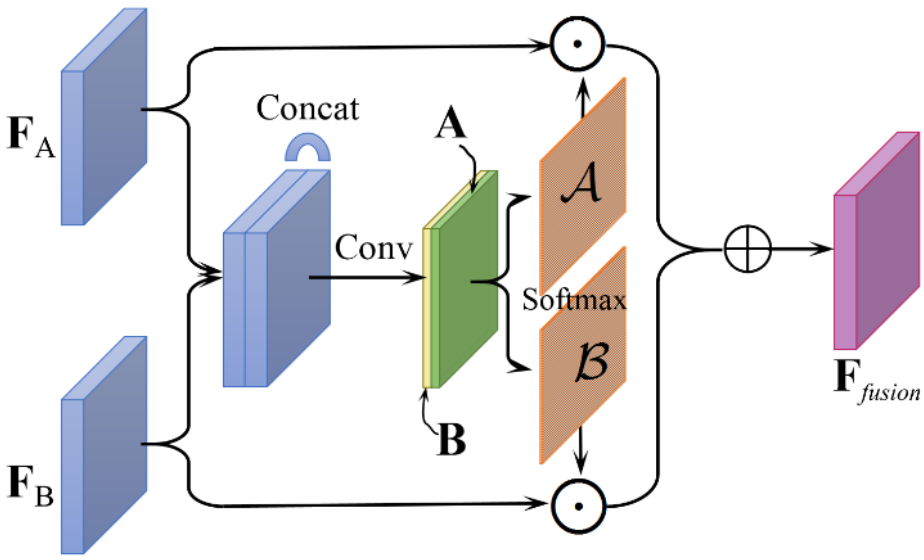
```
class GPG_3(nn.Module):
    def __init__(self, in_channels, width=512, up_kwargs=None,
norm_layer=nn.BatchNorm2d):
```

```
        super(GPG_3, self).__init__()
        self.up_kwargs = up_kwargs

        self.conv5 = nn.Sequential(
            nn.Conv2d(in_channels[-1], width, 3, padding=1, bias=False),
            nn.BatchNorm2d(width),
            nn.ReLU(inplace=True))
        self.conv4 = nn.Sequential(
            nn.Conv2d(in_channels[-2], width, 3, padding=1, bias=False),
            nn.BatchNorm2d(width),
            nn.ReLU(inplace=True))
        self.conv3 = nn.Sequential(
            nn.Conv2d(in_channels[-3], width, 3, padding=1, bias=False),
            nn.BatchNorm2d(width),
            nn.ReLU(inplace=True))
        self.conv_out = nn.Sequential(
            nn.Conv2d(3 * width, width, 1, padding=0, bias=False),
            nn.BatchNorm2d(width))

        self.dilation1 = nn.Sequential(
            SeparableConv2d(3 * width, width, kernel_size=3, padding=1,
dilation=1, bias=False),
            nn.BatchNorm2d(width),
            nn.ReLU(inplace=True))
        self.dilation2 = nn.Sequential(
            SeparableConv2d(3 * width, width, kernel_size=3, padding=2,
dilation=2, bias=False),
            nn.BatchNorm2d(width),
            nn.ReLU(inplace=True))
        self.dilation3 = nn.Sequential(
            SeparableConv2d(3 * width, width, kernel_size=3, padding=4,
dilation=4, bias=False),
            nn.BatchNorm2d(width),
            nn.ReLU(inplace=True))
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_uniform_(m.weight.data)
                if m.bias is not None:
                    m.bias.data.zero_()
            elif isinstance(m, nn.BatchNorm2d):
                init.normal_(m.weight.data, 1.0, 0.02)
                init.constant_(m.bias.data, 0.0)

    def forward(self, *inputs):
        feats = [self.conv5(inputs[-1]), self.conv4(inputs[-2]),
self.conv3(inputs[-3])]
        _, _, h, w = feats[-1].size()
        feats[-2] = F.interpolate(feats[-2], (h, w), **self.up_kwargs)
        feats[-3] = F.interpolate(feats[-3], (h, w), **self.up_kwargs)
        feat = torch.cat(feats, dim=1)
        feat = torch.cat([self.dilation1(feat), self.dilation2(feat),
self.dilation3(feat)], dim=1)
        feat = self.conv_out(feat)
        return feat
```

- SAPF（Scale-Aware Pyramid Fusion Module）



该模块尝试解决在最高层语义信息层有效融合多尺度信息，使用三个不同扩张率的空洞卷积来获取不同比例感受野的信息，不同扩张卷积由共享的权值，目的是为了防止过拟合和减少参数。并设计了scale-aware（尺度感知）模块（一种空间注意力机制）来融合不同比例的功能，空间注意机制来**动态选择合适的尺度特征，并通过自学习对它们进行融合。**最后利用可学习参数a的残差连接得到最终结果。



该空间注意力机制可以动态选择合适的比例特征并通过自学习将其融合，经过concat和卷积后再由softmax得到A,B两个概率谱，再和原先特征值进行点积得到有效特征，最后求和得到融合后的特征图。

$$\mathcal{A}_i = \frac{e^{\mathbf{A}_i}}{e^{\mathbf{B}_i} + e^{\mathbf{A}_i}}, \quad \mathcal{B}_i = \frac{e^{\mathbf{B}_i}}{e^{\mathbf{B}_i} + e^{\mathbf{A}_i}}, \quad i = [1, 2, 3 \ldots, H \times W]$$

$$\mathbf{F}_{fusion} = \mathcal{A} \odot \mathbf{F}_A + \mathcal{B} \odot \mathbf{F}_B$$

这一注意力机制相对于CENet中将不同感受野特征信息相加再进一步扩展，将不同感受野有效信息权重加大，性能上也进一步提升。

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

```
    def __init__(self, in_channels):
        super(SAPblock, self).__init__()
        self.conv3x3 = nn.Conv2d(in_channels=in_channels,
out_channels=in_channels, dilation=1, kernel_size=3,padding=1)

        self.bn = nn.ModuleList([nn.BatchNorm2d(in_channels),
nn.BatchNorm2d(in_channels), nn.BatchNorm2d(in_channels)])
        self.conv1x1 = nn.ModuleList(
            [nn.Conv2d(in_channels=2 * in_channels, out_channels=in_channels,
dilation=1, kernel_size=1, padding=0),
             nn.Conv2d(in_channels=2 * in_channels, out_channels=in_channels,
dilation=1, kernel_size=1, padding=0)])
        self.conv3x3_1 = nn.ModuleList(
            [nn.Conv2d(in_channels=in_channels, out_channels=in_channels // 2,
dilation=1, kernel_size=3, padding=1),
             nn.Conv2d(in_channels=in_channels, out_channels=in_channels // 2,
dilation=1, kernel_size=3, padding=1)])
        self.conv3x3_2 = nn.ModuleList(
            [nn.Conv2d(in_channels=in_channels // 2, out_channels=2, dilation=1,
kernel_size=3, padding=1),
             nn.Conv2d(in_channels=in_channels // 2, out_channels=2, dilation=1,
kernel_size=3, padding=1)])
        self.conv_last = ConvBnRelu(in_planes=in_channels,
out_planes=in_channels, ksize=1, stride=1, pad=0, dilation=1)

        self.gamma = nn.Parameter(torch.zeros(1))

        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
        x_size = x.size()
        branches_1 = self.conv3x3(x)
        branches_1 = self.bn[0](branches_1)
#torch.nn.functional.conv2d(input, weight, bias=None, stride=1, padding=0,
dilation=1, groups=1) → Tensor
        branches_2 = F.conv2d(x, self.conv3x3.weight, padding=2, dilation=2)  #
share weight
        branches_2 = self.bn[1](branches_2)

        branches_3 = F.conv2d(x, self.conv3x3.weight, padding=4, dilation=4)  #
share weight
        branches_3 = self.bn[2](branches_3)
        #拼接前两个特征，通道数变为原来的2倍
        feat = torch.cat([branches_1, branches_2], dim=1)
        # feat=feat_cat.detach()
        #拼接后先进行1x1卷积，通道数减半
        feat = self.relu(self.conv1x1[0](feat))
        #再进行3x3卷积，padding为1，通道数再减半
        feat = self.relu(self.conv3x3_1[0](feat))
        #再进行3x3卷积，padding为1，通道数变为2
        att = self.conv3x3_2[0](feat)
        att = F.softmax(att, dim=1)

        att_1 = att[:, 0, :, :].unsqueeze(1)
        att_2 = att[:, 1, :, :].unsqueeze(1)

        fusion_1_2 = att_1 * branches_1 + att_2 * branches_2
```

```
        feat1 = torch.cat([fusion_1_2, branches_3], dim=1)
        # feat=feat_cat.detach()
        feat1 = self.relu(self.conv1x1[0](feat1))
        feat1 = self.relu(self.conv3x3_1[0](feat1))
        att1 = self.conv3x3_2[0](feat1)
        att1 = F.softmax(att1, dim=1)

        att_1_2 = att1[:, 0, :, :].unsqueeze(1)
        att_3 = att1[:, 1, :, :].unsqueeze(1)

        ax = self.relu(self.gamma * (att_1_2 * fusion_1_2 + att_3 * branches_3)
+ (1 - self.gamma) * x)
        ax = self.conv_last(ax)
        return ax
```

下一周

接着复现经典模型，熟悉pytorch，打好基础。

消化关于注意力机制的论文，关于目标检测的理论内容多看看。

探索数据集。