深度学习语义分割理论与实战指南

机器学习实验室

专注于机器学习与深度学习技术

目　录

深度学习语义分割理论与实战指南

A Theory and Practical Guide to Deep Learning Semantic Segmentation

v1.0 louwill

Machine Learning Lab

图像分类、目标检测和图像分割是基于深度学习的计算机视觉三大核心任务。三大任务之间明显存在着一种递进的层级关系，图像分类聚焦于整张图像，目标检测定位于图像具体区域，而图像分割则是细化到每一个像素。基于深度学习的图像分割具体包括语义分割、实例分割和全景分割。语义分割的目的是要给每个像素赋予一个语义标签。语义分割在自动驾驶、场景解析、卫星遥感图像和医学影像等领域都有着广泛的应用前景。本文作为基于PyTorch的语义分割技术手册，对语义分割的基本技术框架、主要网络模型和技术方法提供一个实战性指导和参考。

1. 语义分割概述

1.1 语义分割任务描述

图像分割主要包括语义分割（Semantic Segmentation）和实例分割（Instance Segmentation）。那语义分割和实例分割具体都是什么含义？二者又有什么区别和联系？语义分割是对图像中的每个像素都划分出对应的类别，即实现像素级别的分类；而类的具体对象，即为实例，那么实例分割不但要进行像素级别的分类，还需在具体的类别基础上区别开不同的个体。例如，图像有多个人甲、乙、丙，那边他们的语义分割结果都是人，而实例分割结果却是不同的对象。另外，为了同时实现实例分割与不可数类别的语义分割，相关研究又提出了全景分割（Panoptic Segmentation）的概念。语义分割、实例分割和全景分割具体如图1（b）、（c）和（d）图所示。

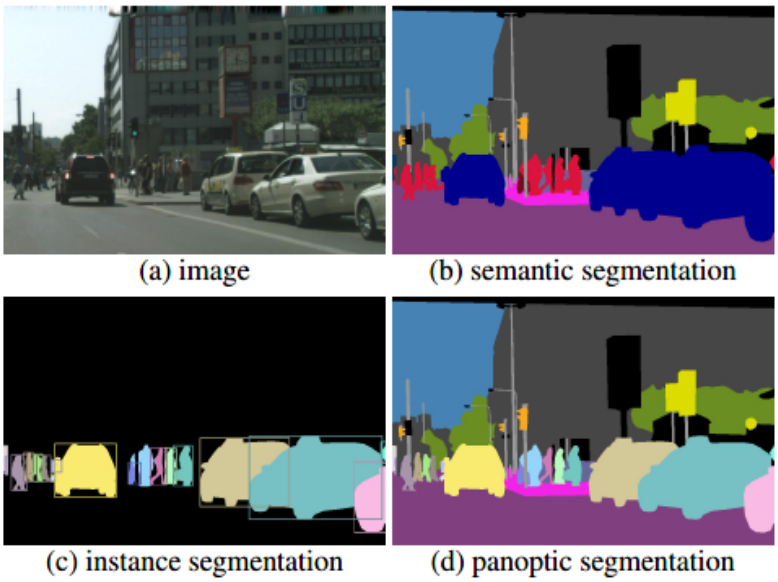


Fig1. Image Segmentation

在开始图像分割的学习和尝试之前，我们必须明确语义分割的任务描述，即搞清楚语义分割的输入输出都是什么。输入是一张原始的RGB图像或者单通道图像，但是输出不再是简单的分类类别或者目标定位，而是带有各个像素类别标签的与输入同分辨率的分割图像。简单来说，我们的输入输出都是图像，而且是同样大小的图像。如图2所示。

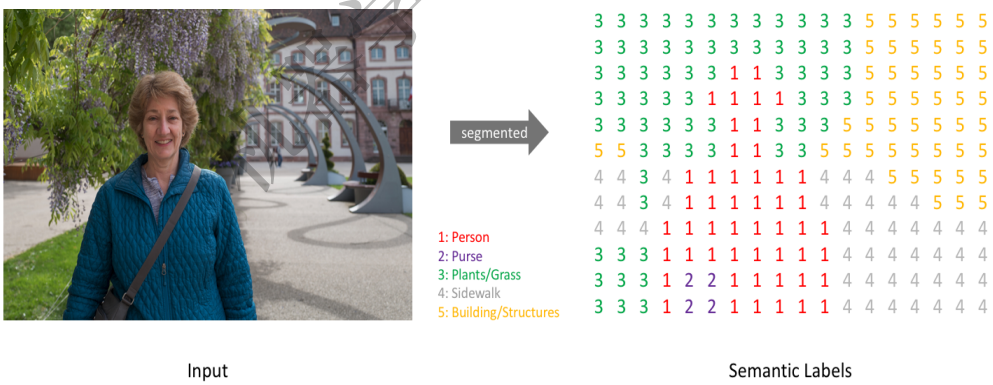


Fig2. Pixel Representation

类似于处理分类标签数据，对预测分类目标采用像素上的one-hot编码，即为每个分类类别创建一个输出的通道。如图3所示。

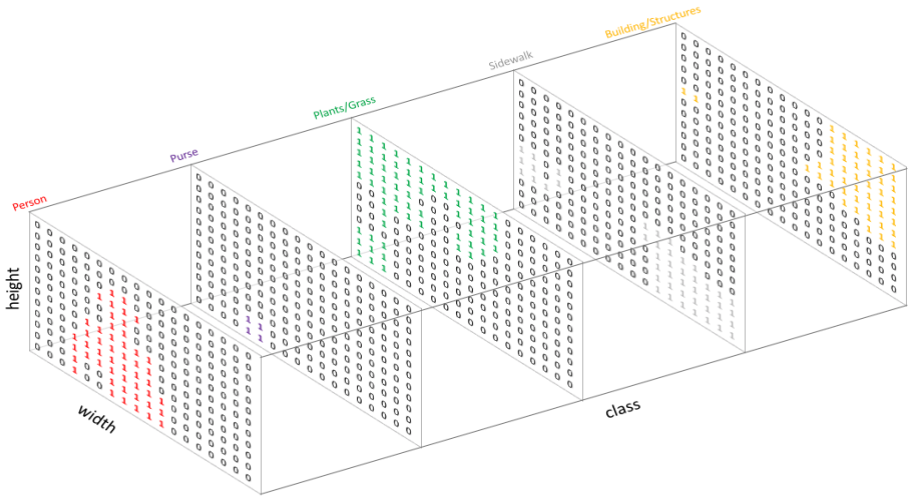


Fig3. Pixel One-hot

图4是将分割图添加到原始图像上的叠加效果。这里需要明确一下mask的概念，在图像处理中我们将其译为掩码，如Mask R-CNN中的Mask。Mask可以理解为我们将预测结果叠加到单个通道时得到的该分类所在区域。

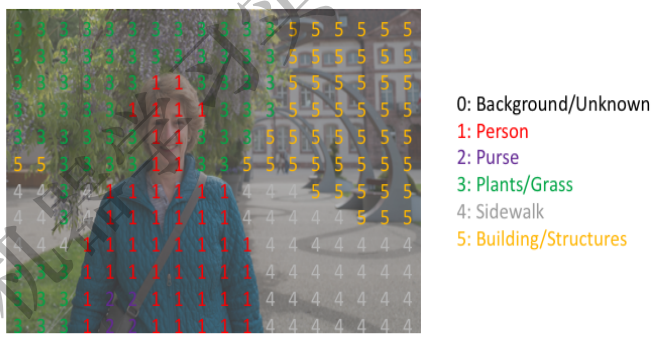


Fig4. Pixel labeling

所以，语义分割的任务就是输入图像经过深度学习算法处理得到带有语义标签的同样尺寸的输出图像。

2. 关键技术组件

在语义分割发展早期，为了能够让深度学习进行像素级的分类任务，在分类任务的基础上对CNN做了一些修改，将分类网络中浓缩语义表征的全连接层去掉，提出用全卷积网络（Fully Convolutional Networks）来处理语义分割问题。然后U-Net的提出，奠定了编解码结构的U形网络深度学习语义分割中的总统山地位。这里我们对语义分割的关键技术组件进行分开描述，编码器、解码器和Skip Connection属于分割网络的核心结构组件，空洞卷积（Dilate Conv）是独立于U形结构的第二大核心设计。条件随机场（CRF）和马尔科夫随机场（MRF）则是用于优化神经网络分割后的细节处理。深监督作为一种常用的结构设计Trick，在分割网络中也有广泛应用。除此之外，则是针对于语义分割的通用技术点。

2.1 编码器与分类网络

编码器对于分割网络来说就是进行特征提取和语义信息浓缩的过程，这对熟悉各种分类网络的我们来说并不陌生。编码器通过卷积和池化的组合不断对图像进行下采样，得到的特征图空间尺寸也会越来越小，但会更加具备语义分辨性。这也是大多数分类网络的通用模式，不断卷积池化使得特征图越来越小，然后配上几层全连接网络即可进行分类判别。常用的分类网络包括AlexNet、VGG、ResNet、Inception、DenseNet和MobileNet等等。

既然之前有那么多优秀的SOTA网络用来做特征提取，所以很多时候分割网络的编码器并不需要我们write from scratch，时刻要有迁移学习的敏感度，直接用现成分类网络的卷积层部分作为编码器进行特征提取和信息浓缩，往往要比从头开始训练一个编码器要快很多。

比如我们以VGG16作为SegNet编码器的预训练模型，以PyTorch为例，来看编码器的写法。

```
from torchvision import models

class SegNet(nn.Module):

    def __init__(self, classes):
        super().__init__()
        vgg16 = models.vgg16(pretrained=True)
        features = vgg16.features
        self.enc1 = features[0: 4]
        self.enc2 = features[5: 9]
        self.enc3 = features[10: 16]
        self.enc4 = features[17: 23]
        self.enc5 = features[24: -1]
```


在上述代码中，可以看到我们将vgg16的31个层分作5个编码模块，每个编码模块的基本结构如下所示：

```
(0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(1): ReLU(inplace=True)
(2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(3): ReLU(inplace=True)
(4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
```

2.2 解码器与上采样

编码器不断将输入不断进行下采样达到信息浓缩，而解码器则负责上采样来恢复输入尺寸。解码器中除了一些卷积方法用为辅助之外，最关键的还是一些上采样方法，主要包括双线性插值、转置卷积和反池化。

双线性插值

插值法（Interpolation）是一种经典的数值分析方法，一些经典插值大家或多或少都有听到过，比如线性插值、三次样条插值和拉格朗日插值法等。在说双线性插值前我们先来了解一下什么是线性插值（Linear interpolation）。线性插值法是指使用连接两个已知量的直线来确定在这两个已知量之间的一个未知量的值的方法。如下图所示：

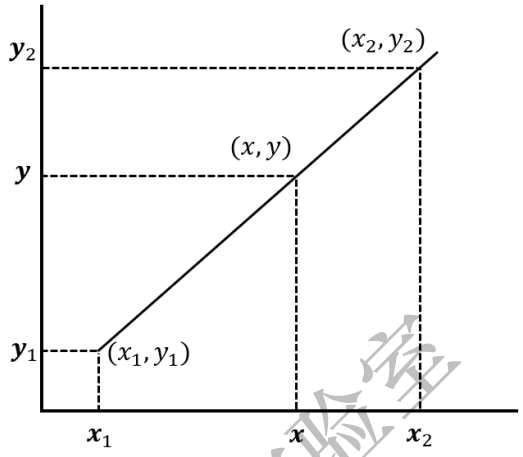


Fig5. Linear Interpolation

已知直线上两点坐标分别为(x_1, y_1)和(x_2, y_2)，现在想要通过线性插值法来得到某一点x在直线上的值。基本就是一个初中数学题，这里就不做过多展开，点x在直线上的值y可以表示为：

$$y = \frac{x_2 - x}{x_2 - x_1} y_2 + \frac{x - x_1}{x_2 - x_1} y_1$$

再来看双线性插值。线性插值用到两个点来确定插值，双线性插值则需要四个点。在图像上采样中，双线性插值利用四个点的像素值来确定要插值的一个像素值，其本质上还是分别在x和y方向上分别进行两次线性插值。如下图所示，我们来看具体做法。

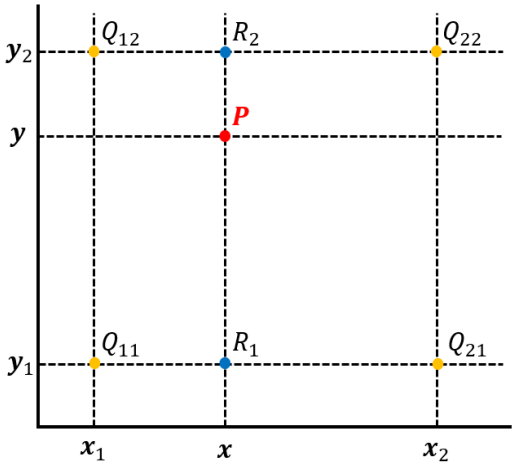


Fig6. Bilinear Interpolation

图中$Q_{11}-Q_{22}$四个黄色的点是已知数据点，红色点P是待插值点。假设Q_{11}为(x_1,y_1)，Q_{12}为(x_1,y_2)，Q_{21}为(x_2,y_1)，Q_{22}为(x_2,y_2)。我们先在x轴方向上进行线性插值，先求得R_1和R_2的插值。根据线性插值公式，有：

$$f(R_1)=\frac{x_2-x}{x_2-x_1}f(Q_{11})+\frac{x-x_1}{x_2-x_1}f(Q_{21})$$
$$f(R_2)=\frac{x_2-x}{x_2-x_1}f(Q_{12})+\frac{x-x_1}{x_2-x_1}f(Q_{22})$$

得到R_1和R_2点坐标之后，便可继续在y轴方向进行线性插值。可得目标点P的插值为：

$$f(P)=\frac{y_2-y}{y_2-y_1}f(R_1)+\frac{y-y_1}{y_2-y_1}f(R_2)$$

双线性插值在众多经典的语义分割网络中都有用到，比如说奠定语义分割编解码框架的FCN网络。假设将3×6的图像通过双线性插值变为6×12的图像，如下图所示。

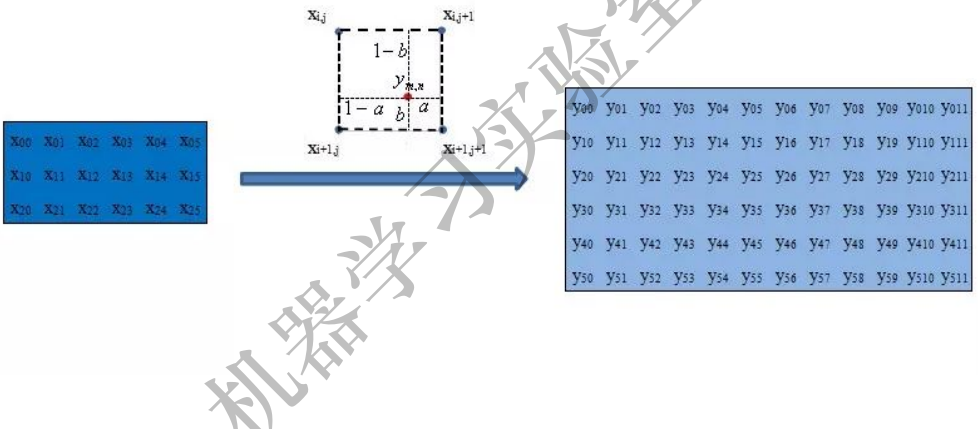


Fig7. Bilinear Interpolation Example

双线性插值的优点是速度非常快，计算量小，但缺点就是效果不是特别理想。

转置卷积

转置卷积（Transposed Convolution）也叫解卷积（Deconvolution），有些人也将其称为反卷积，但这个叫法并不太准确。大家都知道，在常规卷积时，我们每次得到的卷积特征图尺寸是越来越小的。但在图像分割等领域，我们是需要逐步恢复输入时的尺寸的。如果把常规卷积时的特征图不断变小叫做下采样，那么通过转置卷积来恢复分辨率的操作可以称作上采样。

本质上来说，转置卷积跟常规卷积并无区别。不同之处在于先按照一定的比例进行padding来扩大输入尺寸，然后把常规卷积中的卷积核进行转置，再按常规卷积方法进行卷积就是转置卷积。假设输入图像矩阵为X，卷积核矩阵为C，常规卷积的输出为Y，则有：

$$Y=CX$$

两边同时乘以卷积核的转置C^T，这个公式便是转置卷积的输入输出计算。

$$X = C^T Y$$

假设输入大小为4×4，滤波器大小为3×3，常规卷积下输出为2×2，为了演示转置卷积，我们将滤波器矩阵进行稀疏化处理为4×16，将输入矩阵进行拉平为16×1，相应输出结果也会拉平为4×1，图示如下：

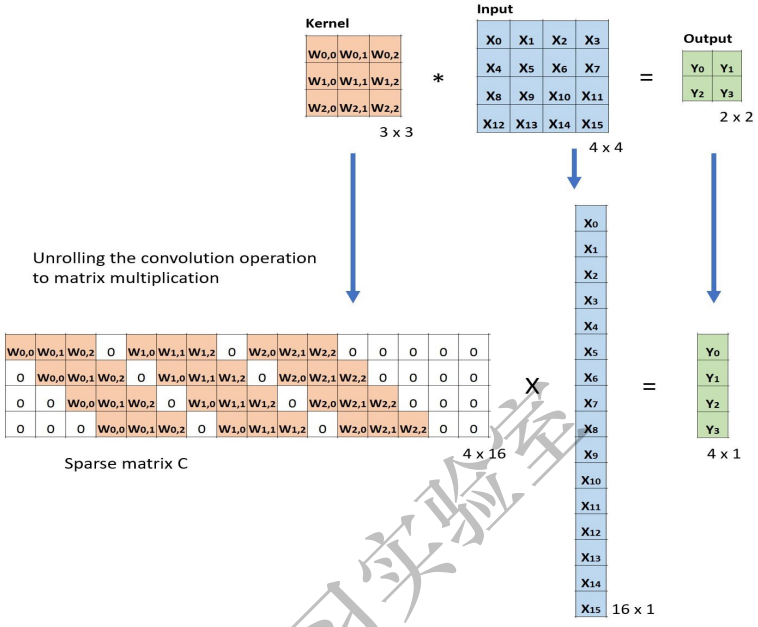


Fig8. Matrix of Convolution

然后按照转置卷积的做法我们把卷积核矩阵进行转置，按照$X = C^T Y$进行验证：

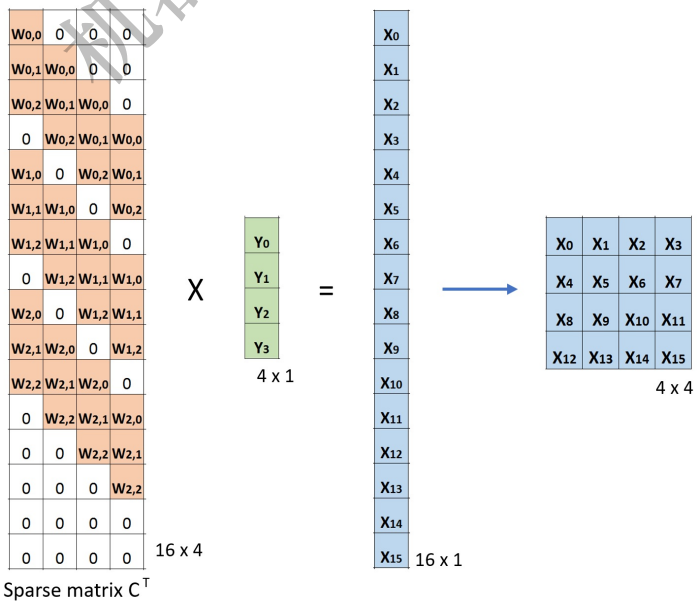


Fig9. Matrix of Transpose Convolution

反池化

反池化（Unpooling）可以理解为池化的逆操作，相较于前两种上采样方法，反池化用的并不是特别多。其简要原理如下，在池化时记录下对应kernel中的坐标，在反池化时将一个元素根据kernel进行放大，根据之前的坐标将元素填写进去，其他位置补位为0即可。

2.3 Skip Connection

跳跃连接本身是在ResNet中率先提出，用于学习一个恒等式和残差结构，后面在DenseNet、FCN和U-Net等网络中广泛使用。最典型的就是U-Net的跳跃连接，在每个编码和解码层之间各添加一个跳跃连接，每一次下采样都会有一个跳跃连接与对应的上采样进行级联，这种不同尺度的特征融合对上采样恢复像素大有帮助。

2.4 Dilate Conv与多尺度

空洞卷积（Dilated/Atrous Convolution）也叫扩张卷积或者膨胀卷积，字面意思上来说就是在卷积核中插入空洞，起到扩大感受野的作用。空洞卷积的直接做法是在常规卷积核中填充0，用来扩大感受野，且进行计算时，空洞卷积中实际只有非零的元素起了作用。假设以一个变量a来衡量空洞卷积的扩张系数，则加入空洞之后的实际卷积核尺寸与原始卷积核尺寸之间的关系：

$$K = k + (k-1)(a-1)$$

其中k为原始卷积核大小，a为卷积扩张率（dilation rate），K为经过扩展后实际卷积核大小。除此之外，空洞卷积的卷积方式跟常规卷积一样。当$a=1$时，空洞卷积就退化为常规卷积。$a=1,2,4$时，空洞卷积示意图如下：

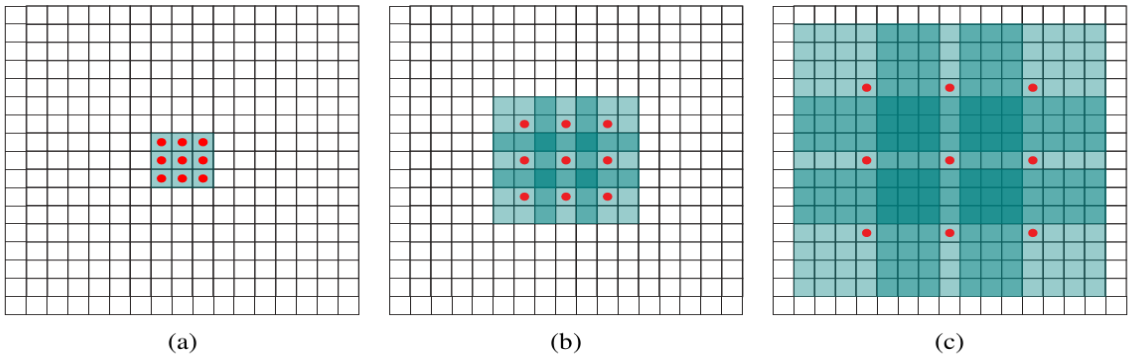


Fig10. Dialate Convolution

当$a=1$，原始卷积核size为3×3，就是常规卷积。$a=2$时，加入空洞之后的卷积核$size=3+(3-1)\times(2-1)=5$，对应的感受野可计算为$2^{(a+2)}-1=7$。$a=3$时，卷积核size可以变化到$3+(3-1)(4-1)=9$，感受野则增长到$2^{(a+2)}-1=15$。对比不加空洞卷积的情况，在stride为1的情况下3层3x3卷积的叠加，第三层输出特征图对应的感受野也只有$1+(3-1)\times 3=7$。所以，空洞卷积的一个重要作用就是增大感受野。

在语义分割的发展历程中，增大感受野是一个非常重要的设计。早期FCN提出以全卷积方式来处理像素级别的分割任务时，包括后来奠定语义分割baseline地位的U-Net，网络结构中存在大量的池化层来进行下采样，大量使用池化层的结果就是损失掉了一些信息，在解码上采样重建分辨率的时候肯定会有影响。特别是对于多目标、小物体的语义分割问题，以U-Net为代表的分割模型一直存在着精度瓶颈的问题。而基于增大感受野的动机背景下就提出了以空洞卷积为重大创新的deeplab系列分割网络，我们在深度学习语义分割模型中会对deeplab进行详述，这里不做过多展开。

对于语义分割而言，空洞卷积主要有三个作用：

- 第一是扩大感受野，具体前面已经说的比较多了，这里不做重复。但需要明确一点，池化也可以扩大感受野，但空间分辨率降低了，相比之下，空洞卷积可以在扩大感受野的同时不丢失分辨率，且保持像素的相对空间位置不变。简单而言就是空洞卷积可以同时控制感受野和分辨率。
- 第二就是获取多尺度上下文信息。当多个带有不同dilation rate的空洞卷积核叠加时，不同的感受野会带来多尺度信息，这对于分割任务是非常重要的。
- 第三就是可以降低计算量，不需要引入额外的参数，如上图空洞卷积示意图所示，实际卷积时只有带有红点的元素真正进行计算。

2.5 后处理技术

早期语义分割模型效果较为粗糙，在没有更好的特征提取模型的情况下，研究者们便在神经网络模型的粗糙结果进行后处理（Post-Processing），主要方法就是一些常用的概率图模型，比如说条件随机场（Conditional Random Field,CRF）和马尔可夫随机场（Markov Random Field,MRF）。
CRF是一种经典的概率图模型，简单而言就是给定一组输入序列的条件下，求另一组输出序列的条件概率分布模型，CRF在自然语言处理领域有着广泛应用。CRF在语义分割后处理中用法的基本思路如下：对于FCN或者其他分割网络的粗粒度分割结果而言，每个像素点i具有对应的类别标签x_i和观测值y_i，以每个像素为节点，以像素与像素之间的关系作为边即可构建一个CRF模型。在这个CRF模型中，我们通过观测变量y_i来预测像素i对应的标签值x_i。

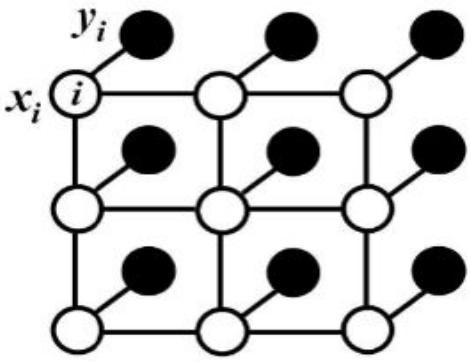


Fig11. CRF

以上做法也叫DenseCRF，具体细节可参考论文：
Efficient Inference in Fully Connected CRFs with Gaussian Edge Potentials，除此之外还有

CRFasRNN，采用平均场近似的方式将CRF方法融入到神经网络过程中，本质上是对DenseCRF的一种优化。

另一种后处理概率图模型是MRF，MRF与CRF较为类似，只是对CRF的二元势函数做了调整，其优点在于可以使用平均场来构造CNN网络，并且推理过程可以一次性搞定。MRF在Deep Parsing Network(DPN)中有详细描述，相关细节可参考论文Semantic Image Segmentation via Deep Parsing Network。

语义分割发展前期，在分割网络模型的结果上加上CRF和MRF等后处理技术形成了早期的语义分割技术框架：

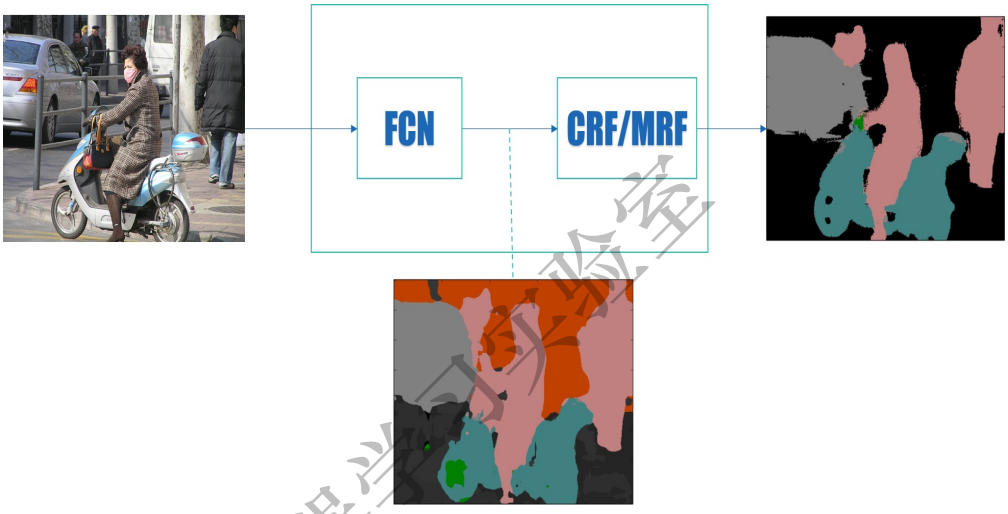


Fig12. Framework of Semantic Segmentation with CRF/MRF

但从Deeplab v3开始，主流的语义分割网络就不再热衷于后处理技术了。一个典型的观点认为神经网络分割效果不好才会用后处理技术，这说明在分割网络本身上还有很大的提升空间。一是CRF本身不太容易训练，二来语义分割任务的端到端趋势。后来语义分割领域的SOTA网络也确实证明了这一点。尽管如此，CRF等后处理技术作为语义分割发展历程上的一个重要方法，我们有必要在此进行说明。从另一方面看，深度学习和概率图的结合虽然并不是那么顺利，但相信未来依旧会大有前景。

2.6 深监督

所谓深监督（Deep Supervision），就是在深度神经网络的某些中间隐藏层加了一个辅助的分类器作为一种网络分支来对主干网络进行监督的技巧，用来解决深度神经网络训练梯度消失和收敛速度过慢等问题。

带有深监督的一个8层深度卷积网络结构如下图所示。

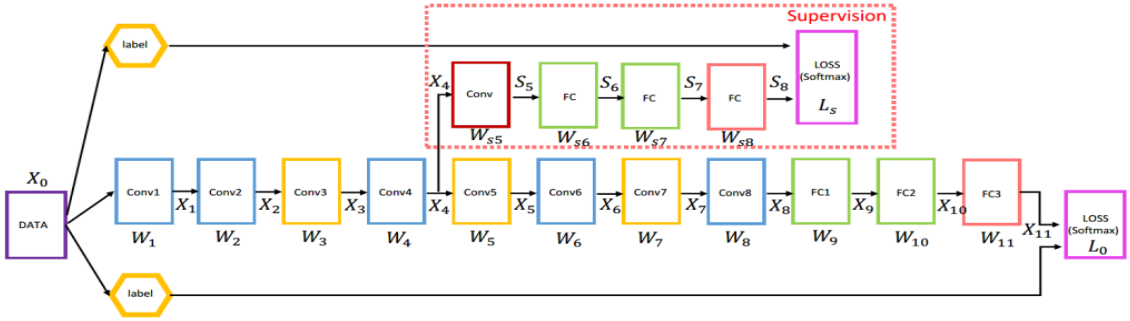


Fig13. Deep Supervision Example

可以看到，图中在第四个卷积块之后添加了一个监督分类器作为分支。 Conv4 输出的特征图除了随着主网络进入 Conv5 之外，也作为输入进入了分支分类器。如图所示，该分支分类器包括一个卷积块、两个带有 Dropout 和 ReLu 的全连接块和一个纯全连接块。带有深监督的卷积模块例子如下。

```
class C1DeepSup(nn.Module):
    def __init__(self, num_class=150, fc_dim=2048, use_softmax=False):
        super(C1DeepSup, self).__init__()
        self.use_softmax = use_softmax
        self.cbr = conv3x3_bn_relu(fc_dim, fc_dim // 4, 1)
        self.cbr_deepsup = conv3x3_bn_relu(fc_dim // 2, fc_dim // 4, 1)
        # 最后一层卷积
        self.conv_last = nn.Conv2d(fc_dim // 4, num_class, 1, 1, 0)
        self.conv_last_deepsup = nn.Conv2d(fc_dim // 4, num_class, 1, 1, 0)

    # 前向计算流程
    def forward(self, conv_out, segSize=None):
        conv5 = conv_out[-1]
        x = self.cbr(conv5)
        x = self.conv_last(x)
        # is True during inference
        if self.use_softmax:
            x = nn.functional.interpolate(
                x, size=segSize, mode='bilinear', align_corners=False)
            x = nn.functional.softmax(x, dim=1)
            return x
        # 深监督模块
        conv4 = conv_out[-2]
        _ = self.cbr_deepsup(conv4)
        _ = self.conv_last_deepsup(_)

        # 主干卷积网络softmax输出
        x = nn.functional.log_softmax(x, dim=1)
        # 深监督分支网络softmax输出
        _ = nn.functional.log_softmax(_, dim=1)
        return (x, _)
```

2.7 通用技术

通用技术主要是指深度学习流程中会用到的基本模块，比如说损失函数的选取以及采用哪种精度衡量指标。其他的像优化器的选取，学习率的控制等，这里限于篇幅进行省略。

损失函数

常用的分类损失均可用作语义分割的损失函数。最常用的就是交叉熵损失函数，如果只是前景分割，则可以使用二分类的交叉熵损失（Binary CrossEntropy Loss, BCE loss），对于目标物体较小的情况我们可以使用Dice损失，对于目标物体类别不均衡的情况可以使用加权的交叉熵损失（Weighted CrossEntropy Loss, WCE Loss），另外也可以尝试多种损失函数的组合。

精度描述

语义分割作为经典的图像分割问题，其本质上还是一种图像像素分类。既然是分类，我们就可以使用常见的分类评价指标来评估模型好坏。语义分割常见的评价指标包括像素准确率（Pixel Accuracy）、平均像素准确率（Mean Pixel Accuracy）、平均交并比（Mean IoU）、频权交并比（FWIoU）和Dice系数（Dice Coeffcient）等。

像素准确率(PA)。 像素准确率跟分类中的准确率含义一样，即所有分类正确的像素数占全部像素的比例。PA的计算公式如下：

$$PA = \frac{\sum_{i=0}^{n} p_{ii}}{\sum_{i=0}^{n}\sum_{j=0}^{n} p_{ij}}$$

平均像素准确率(MPA)。 平均像素准确率其实更应该叫平均像素精确率，是指分别计算每个类别分类正确的像素数占所有预测为该类别像素数比例的平均值。所以，从定义上看，这是精确率(Precision)的定义，MPA的计算公式如下：

$$MPA = \frac{1}{n+1}\sum_{i=0}^{n} \frac{p_{ii}}{\sum_{j=0}^{n} p_{ij}}$$

平均交并比(MIoU)。 交并比（Intersection over Union）的定义很简单，将标签图像和预测图像看成是两个集合，计算两个集合的交集和并集的比值。而平均交并比则是将所有类的IoU取平均。 MIoU的计算公式如下：

$$MIoU = \frac{1}{n+1}\sum_{i=0}^{n} \frac{p_{ii}}{\sum_{j=0}^{n} p_{ij} + \sum_{j=0}^{n} p_{ji} - p_{ii}}$$

频权交并比(FWIoU)。 频权交并比顾名思义，就是以每一类别的频率为权重和其IoU加权计算出来的结果。FWIoU的设计思想很明确，语义分割很多时候会面临图像中各目标类别不平衡的情况，对各类别IoU直接求平均不是很合理，所以考虑各类别的权重就非常重要了。FWIoU的计算公式如下：

$$FWIoU = \frac{1}{\sum_{i=0}^{n}\sum_{j=0}^{n} p_{ij}}\sum_{i=0}^{n} \frac{\sum_{j=0}^{n} p_{ij} p_{ii}}{\sum_{j=0}^{n} p_{ij} + \sum_{j=0}^{n} p_{ji} - p_{ii}}$$

Dice系数。 Dice系数是一种度量两个集合相似性的函数，是语义分割中最常用的评价指标之一。Dice系数定义为两倍的交集除以像素和，跟IoU有点类似，其计算公式如下：

$$dice = \frac{2|X \cap Y|}{|X|+|Y|}$$

dice本质上跟分类指标中的F1-Score类似。作为最常用的分割指标之一，这里给出PyTorch的实现方式。

```
import torch

def dice_coef(pred, target):
    """
    Dice = (2*|X & Y|)/ (|X|+ |Y|)
         =  2*sum(|A*B|)/(sum(A^2)+sum(B^2))
    """
    smooth = 1.
    m1 = pred.view(-1).float()
    m2 = target.view(-1).float()
    intersection = (m1 * m2).sum().float()
    dice = (2. * intersection + smooth) / (torch.sum(m1*m1) + torch.sum(m2*m2) + smooth)
    return dice
```


3. 数据Pipeline

这里主要说一下PyTorch的自定义数据读取pipeline模板和相关trciks以及如何优化数据读取的pipeline等。我们从PyTorch的数据对象类 `Dataset` 开始。 `Dataset` 在PyTorch中的模块位于 `utils.data` 下。

```
from torch.utils.data import Dataset
```

3.1 Torch数据读取模板

PyTorch官方为我们提供了自定义数据读取的标准化代码代码模块，作为一个读取框架，我们这里称之为原始模板。其代码结构如下：

```
from torch.utils.data import Dataset

class CustomDataset(Dataset):
    def __init__(self, ...):
        # stuff

    def __getitem__(self, index):
        # stuff
        return (img, label)

    def __len__(self):
        # return examples size
        return count
```

3.2 transform与数据增强

PyTorch数据增强功能可以放在 `transform` 模块下，添加 `transform` 后的数据读取结构如下所示：

```
from torch.utils.data import Dataset
from torchvision import transforms as T

class CustomDataset(Dataset):
    def __init__(self, ...):
        # stuff
        # ...
        # compose the transforms methods
        self.transform = T.Compose([T.CenterCrop(100),
                                    T.RandomResizedCrop(256),
                                    T.RandomRotation(45),
                                    T.ToTensor()])

    def __getitem__(self, index):
        # stuff
        # ...
        data = # Some data read from a file or image
        labe = # Some data read from a file or image

        # execute the transform
        data = self.transform(data)
        label = self.transform(label)
        return (img, label)

    def __len__(self):
        # return examples size
        return count

if __name__ == '__main__':
    # Call the dataset
    custom_dataset = CustomDataset(...)
```


需要说明的是，PyTorch `transform` 模块所做的数据增强并不是我们所理解的广义上的数据增强。`transform` 所做的增强，仅仅是在数据读取过程中随机地对某张图像做转化操作，实际数据量上并没有增多，可以将其视为是一种在线增强的策略。如果想要实现实际训练数据成倍数的增加，可以使用离线增强策略。

与图像分类仅需要对输入图像做增强不同的是，对于语义分割的数据增强而言，需要同时对输入图像和输入的mask同步进行数据增强工作。实际写代码时，要记得使用随机种子，在不失随机性的同时，保证输入图像和输出mask具备同样的转换。一个完整的语义分割在线数据增强代码实例如下：

```
import os
import random
import torch
from torch.utils.data import Dataset
from PIL import Image

class SegmentationDataset(Dataset):
    # read the input images
    @staticmethod
    def _load_input_image(path):
        with open(path, 'rb') as f:
            img = Image.open(f)
            return img.convert('RGB')
    # read the mask images
    @staticmethod
    def _load_target_image(path):
        with open(path, 'rb') as f:
            img = Image.open(f)
            return img.convert('L')

    def __init__(self, input_root, target_root, transform_input=None,
                 transform_target=None, seed_fn=None):
        self.input_root = input_root
        self.target_root = target_root
        self.transform_input = transform_input
        self.transform_target = transform_target
        self.seed_fn = seed_fn
        # sort the ids
        self.input_ids = sorted(img for img in os.listdir(self.input_root))
        self.target_ids = sorted(img for img in os.listdir(self.target_root))
        assert(len(self.input_ids) == len(self.target_ids))

    # set random number seed
    def _set_seed(self, seed):
        random.seed(seed)
        torch.manual_seed(seed)
        if self.seed_fn:
            self.seed_fn(seed)

    def __getitem__(self, idx):
        input_img = self._load_input_image(
            os.path.join(self.input_root, self.input_ids[idx]))
        target_img = self._load_target_image(
            os.path.join(self.target_root, self.target_ids[idx]))

        if self.transform_input:
            # ensure that the input and output have the same randomness.
            seed = random.randint(0, 2**32)
            self._set_seed(seed)
            input_img = self.transform_input(input_img)
            self._set_seed(seed)
```

```
            target_img = self.transform_target(target_img)
        return input_img, target_img, self.input_ids[idx]

    def __len__(self):
        return len(self.input_ids)
```

其中 `transform_input` 和 `transform_target` 均可由 `transform` 模块下的函数封装而成。一个皮肤病灶分割的在线数据增强实例效果如下图所示。

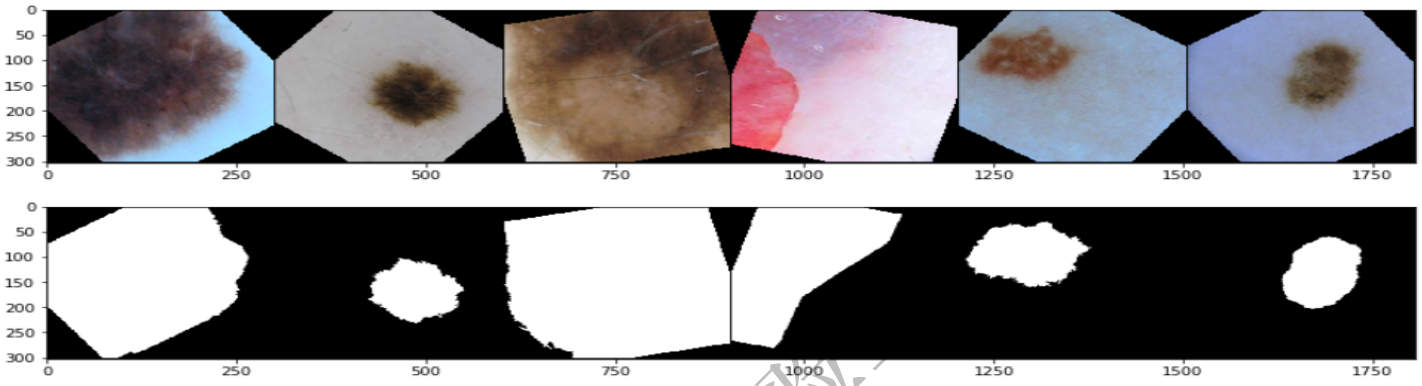


Fig14. Example of online augmentation

4. 模型与算法

早期基于深度学习的图像分割以FCN为核心，旨在重点解决如何更好从卷积下采样中恢复丢掉的信息损失。后来逐渐形成了以U-Net为核心的这样一种编解码对称的U形结构。**语义分割界迄今为止最重要的两个设计，一个是以U-Net为代表的U形结构，目前基于U-Net结构的创新就层出不穷，比如说应用于3D图像的V-Net，嵌套U-Net结构的U-Net++等。除此在外还有SegNet、RefineNet、HRNet和FastFCN。另一个则是以DeepLab系列为代表的Dilation设计，主要包括DeepLab系列和PSPNet。随着模型的Baseline效果不断提升，语义分割任务的主要矛盾也逐从downsample损失恢复像素逐渐演变为如何更有效地利用context上下文信息。**

4.1 FCN

FCN（Fully Convilutional Networks）是语义分割领域的开山之作。FCN的提出是在2016年，相较于此前提出的AlexNet和VGG等卷积全连接的网络结构，FCN提出用卷积层代替全连接层来处理语义分割问题，这也是FCN的由来，即全卷积网络。

FCN的关键点主要有三，一是全卷积进行特征提取和下采样，二是双线性插值进行上采样，三是跳跃连接进行特征融合。

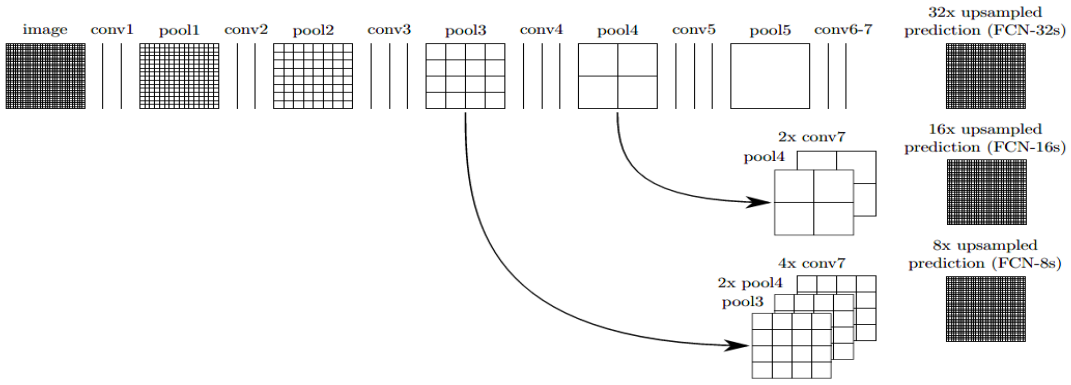


Fig15. FCN

利用PyTorch实现一个FCN-8网络：

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

from torch.utils import model_zoo
from torchvision import models

class FCN8(nn.Module):

    def __init__(self, num_classes):
        super().__init__()

        feats = list(models.vgg16(pretrained=True).features.children())

        self.feats = nn.Sequential(*feats[0:9])
        self.feat3 = nn.Sequential(*feats[10:16])
        self.feat4 = nn.Sequential(*feats[17:23])
        self.feat5 = nn.Sequential(*feats[24:30])

        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                m.requires_grad = False

        self.fconn = nn.Sequential(
            nn.Conv2d(512, 4096, 7),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Conv2d(4096, 4096, 1),
            nn.ReLU(inplace=True),
            nn.Dropout(),
        )
        self.score_feat3 = nn.Conv2d(256, num_classes, 1)
        self.score_feat4 = nn.Conv2d(512, num_classes, 1)
        self.score_fconn = nn.Conv2d(4096, num_classes, 1)

    def forward(self, x):
        feats = self.feats(x)
        feat3 = self.feat3(feats)
        feat4 = self.feat4(feat3)
        feat5 = self.feat5(feat4)
        fconn = self.fconn(feat5)

        score_feat3 = self.score_feat3(feat3)
        score_feat4 = self.score_feat4(feat4)
        score_fconn = self.score_fconn(fconn)

        score = F.upsample_bilinear(score_fconn, score_feat4.size()[2:])
        score += score_feat4
        score = F.upsample_bilinear(score, score_feat3.size()[2:])
        score += score_feat3
```

```
        return F.upsample_bilinear(score, x.size()[2:])
```

从代码中可以看到，我们使用了vgg16作为FCN-8的编码部分，这使得FCN-8具备较强的特征提取能力。

4.2 UNet

早期基于深度学习的图像分割以FCN为核心，旨在重点解决如何更好从卷积下采样中恢复丢掉的信息损失。后来逐渐形成了以UNet为核心的这样一种编解码对称的U形结构。

UNet结构能够在分割界具有一统之势，最根本的还是其效果好，尤其是在医学图像领域。所以，做医学影像相关的深度学习应用时，一定都用过UNet，而且最原始的UNet一般都会有一个不错的baseline表现。2015年发表UNet的MICCAI，是目前医学图像分析领域最顶级的国际会议，UNet为什么在医学上效果这么好非常值得探讨一番。

U-Net结构如下图所示：

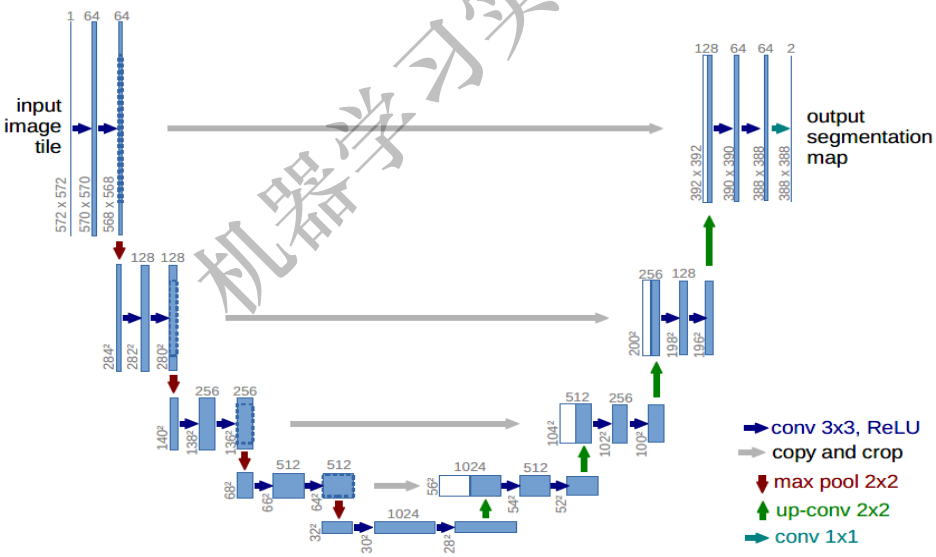


Fig16. UNet

乍一看很复杂，U形结构下貌似有很多细节问题。我们来把UNet简化一下，如下图所示：

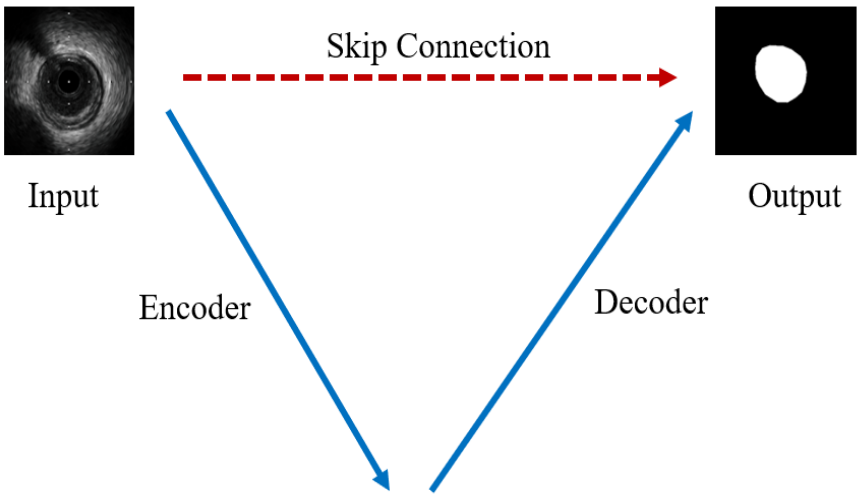


Fig17. U-Net简化

从图中可以看到，简化之后的UNet的关键点只有三条线：

- 下采样编码
- 上采样解码
- 跳跃连接

下采样进行信息浓缩和上采样进行像素恢复，这是其他分割网络都会有的部分，UNet自然也不会跳出这个框架，可以看到，UNet进行了4次的最大池化下采样，每一次采样后都使用了卷积进行信息提取得到特征图，然后再经过4次上采样恢复输入像素尺寸。但UNet最关键的、也是最特色的部分在于图中红色虚线的Skip Connection。每一次下采样都会有一个跳跃连接与对应的上采样进行级联，这种不同尺度的特征融合对上采样恢复像素大有帮助，具体来说就是高层（浅层）下采样倍数小，特征图具备更加细致的图特征，底层（深层）下采样倍数大，信息经过大量浓缩，空间损失大，但有助于目标区域（分类）判断，当high level和low level的特征进行融合时，分割效果往往会非常好。从某种程度上讲，这种跳跃连接也可以视为一种Deep Supervision。

U-Net的简单实现如下：

```
# 编码块
class UNetEnc(nn.Module):

    def __init__(self, in_channels, out_channels, dropout=False):
        super().__init__()

        layers = [
            nn.Conv2d(in_channels, out_channels, 3, dilation=2),
            nn.ReLU(inplace=True),
            nn.Conv2d(out_channels, out_channels, 3, dilation=2),
            nn.ReLU(inplace=True),
        ]
        if dropout:
            layers += [nn.Dropout(.5)]
        layers += [nn.MaxPool2d(2, stride=2, ceil_mode=True)]

        self.down = nn.Sequential(*layers)

    def forward(self, x):
        return self.down(x)

# 解码块
class UNetDec(nn.Module):

    def __init__(self, in_channels, features, out_channels):
        super().__init__()

        self.up = nn.Sequential(
            nn.Conv2d(in_channels, features, 3),
            nn.ReLU(inplace=True),
            nn.Conv2d(features, features, 3),
            nn.ReLU(inplace=True),
            nn.ConvTranspose2d(features, out_channels, 2, stride=2),
            nn.ReLU(inplace=True),
        )

    def forward(self, x):
        return self.up(x)
# U-Net
class UNet(nn.Module):

    def __init__(self, num_classes):
        super().__init__()

        self.enc1 = UNetEnc(3, 64)
        self.enc2 = UNetEnc(64, 128)
        self.enc3 = UNetEnc(128, 256)
        self.enc4 = UNetEnc(256, 512, dropout=True)
        self.center = nn.Sequential(
            nn.Conv2d(512, 1024, 3),
            nn.ReLU(inplace=True),
```

```
            nn.Conv2d(1024, 1024, 3),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.ConvTranspose2d(1024, 512, 2, stride=2),
            nn.ReLU(inplace=True),
        )
        self.dec4 = UNetDec(1024, 512, 256)
        self.dec3 = UNetDec(512, 256, 128)
        self.dec2 = UNetDec(256, 128, 64)
        self.dec1 = nn.Sequential(
            nn.Conv2d(128, 64, 3),
            nn.ReLU(inplace=True),
            nn.Conv2d(64, 64, 3),
            nn.ReLU(inplace=True),
        )
        self.final = nn.Conv2d(64, num_classes, 1)

    # 前向传播过程
    def forward(self, x):
        enc1 = self.enc1(x)
        enc2 = self.enc2(enc1)
        enc3 = self.enc3(enc2)
        enc4 = self.enc4(enc3)
        center = self.center(enc4)
        # 包含了同层分辨率级联的解码块
        dec4 = self.dec4(torch.cat([
            center, F.upsample_bilinear(enc4, center.size()[2:])], 1))
        dec3 = self.dec3(torch.cat([
            dec4, F.upsample_bilinear(enc3, dec4.size()[2:])], 1))
        dec2 = self.dec2(torch.cat([
            dec3, F.upsample_bilinear(enc2, dec3.size()[2:])], 1))
        dec1 = self.dec1(torch.cat([
            dec2, F.upsample_bilinear(enc1, dec2.size()[2:])], 1))

        return F.upsample_bilinear(self.final(dec1), x.size()[2:])
```


4.3 SegNet

SegNet网络是典型的编码-解码结构。SegNet编码器网络由VGG16的前13个卷积层构成，所以通常是使用VGG16的预训练权重来进行初始化。每个编码器层都有一个对应的解码器层，因此解码器层也有13层。编码器最后的输出输入到softmax分类器中，输出每个像素的类别概率。SegNet如下图所示。

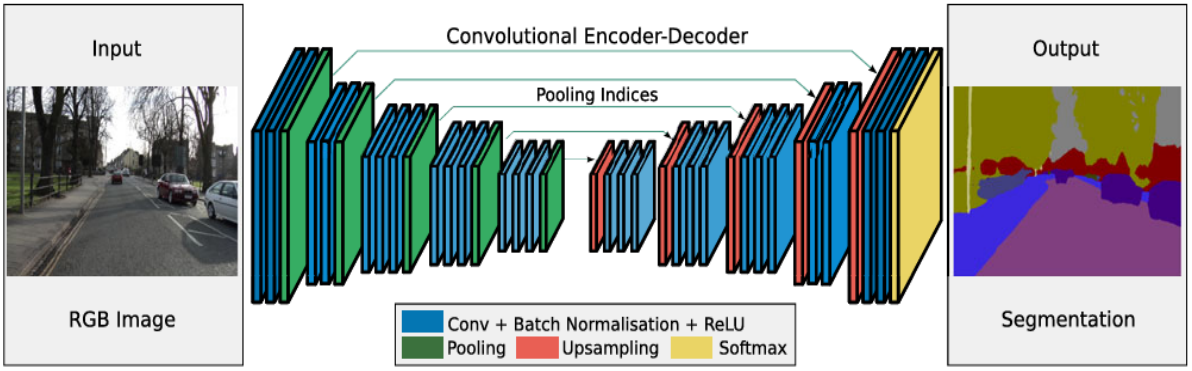


Fig18. SegNet结构

SegNet的一个简易参考实现如下：

```
import torch
import torch.nn as nn
import torch.nn.init as init
import torch.nn.functional as F
from torchvision import models

# define Decoder
class SegNetDec(nn.Module):

    def __init__(self, in_channels, out_channels, num_layers):
        super().__init__()
        layers = [
            nn.Conv2d(in_channels, in_channels // 2, 3, padding=1),
            nn.BatchNorm2d(in_channels // 2),
            nn.ReLU(inplace=True),
        ]
        layers += [
            nn.Conv2d(in_channels // 2, in_channels // 2, 3, padding=1),
            nn.BatchNorm2d(in_channels // 2),
            nn.ReLU(inplace=True),
        ] * num_layers
        layers += [
            nn.Conv2d(in_channels // 2, out_channels, 3, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU(inplace=True),
        ]
        self.decode = nn.Sequential(*layers)

    def forward(self, x):
        return self.decode(x)

# SegNet
class SegNet(nn.Module):

    def __init__(self, classes):
        super().__init__()
        vgg16 = models.vgg16(pretrained=True)
        features = vgg16.features
        self.enc1 = features[0: 4]
        self.enc2 = features[5: 9]
        self.enc3 = features[10: 16]
        self.enc4 = features[17: 23]
        self.enc5 = features[24: -1]

        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                m.requires_grad = False

        self.dec5 = SegNetDec(512, 512, 1)
        self.dec4 = SegNetDec(512, 256, 1)
        self.dec3 = SegNetDec(256, 128, 1)
```

```
        self.dec2 = SegNetDec(128, 64, 0)

        self.final = nn.Sequential(*[
            nn.Conv2d(64, classes, 3, padding=1),
            nn.BatchNorm2d(classes),
            nn.ReLU(inplace=True)
        ])

    def forward(self, x):
        x1 = self.enc1(x)
        e1, m1 = F.max_pool2d(x1, kernel_size=2, stride=2, return_indices=True)
        x2 = self.enc2(e1)
        e2, m2 = F.max_pool2d(x2, kernel_size=2, stride=2, return_indices=True)
        x3 = self.enc3(e2)
        e3, m3 = F.max_pool2d(x3, kernel_size=2, stride=2, return_indices=True)
        x4 = self.enc4(e3)
        e4, m4 = F.max_pool2d(x4, kernel_size=2, stride=2, return_indices=True)
        x5 = self.enc5(e4)
        e5, m5 = F.max_pool2d(x5, kernel_size=2, stride=2, return_indices=True)

        def upsample(d):
            d5 = self.dec5(F.max_unpool2d(d, m5, kernel_size=2, stride=2, output_size=x5.size()))
            d4 = self.dec4(F.max_unpool2d(d5, m4, kernel_size=2, stride=2, output_size=x4.size()))
            d3 = self.dec3(F.max_unpool2d(d4, m3, kernel_size=2, stride=2, output_size=x3.size()))
            d2 = self.dec2(F.max_unpool2d(d3, m2, kernel_size=2, stride=2, output_size=x2.size()))
            d1 = F.max_unpool2d(d2, m1, kernel_size=2, stride=2, output_size=x1.size())
            return d1

        d = upsample(e5)
        return self.final(d)
```


4.4 Deeplab系列

Deeplab系列可以算是深度学习语义分割的另一个主要架构，其代表方法就是基于Dilation的多尺度设计。Deeplab系列主要包括：

- Deeplab v1
- Deeplab v2
- Deeplab v3
- Deeplab v3+

Deeplab v1主要是率先使用了空洞卷积，是Deeplab系列最原始的版本。Deeplab v2在Deeplab v1的基础上最大的改进在于提出了ASPP（Atrous Spatial Pyramid Pooling），即带有空洞卷积的金字塔池化，该设计的主要目的就是提取图像的多尺度特征。另外Deeplab v2也将Deeplab v1的Backone网络更换为ResNet。Deeplab v1和v2还有一个比较大的特点就是使用了CRF作为后处理技术。

这里重点说一下多尺度问题。多尺度问题就是当图像中的目标对象存在不同大小时，分割效果不佳的现象。比如同样的物体，在近处拍摄时物体显得大，远处拍摄时显得小。解决多尺度问题的目标就是不论目标对象是大还是小，网络都能将其分割地很好。Deeplab v2使用ASPP处理多尺度问题，ASPP设计结构如下图所示。

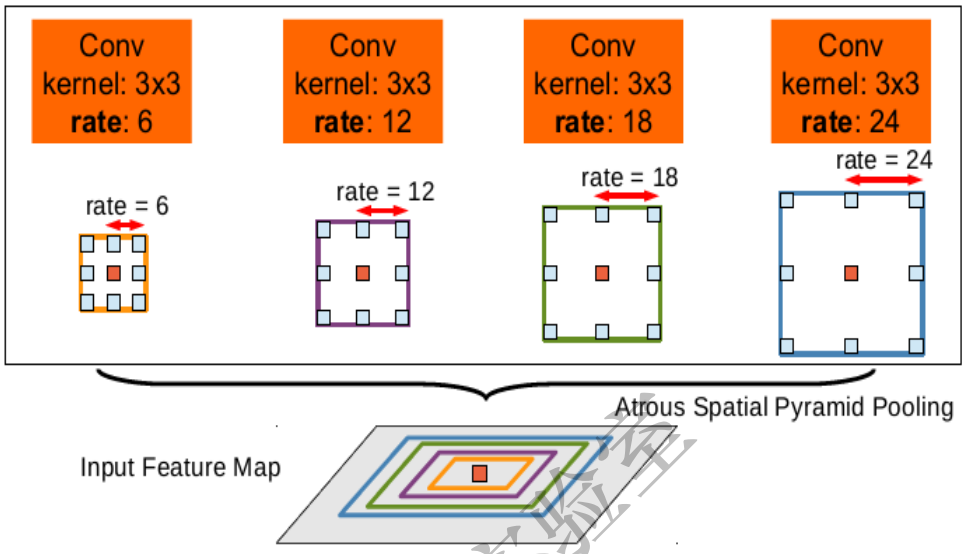


Fig19. ASPP

从Deeplab v3开始，Deeplab系列舍弃了CRF后处理模块，提出了更加通用的、适用任何网络的分割框架，对ResNet最后的Block做了复制和级联（Cascade），对ASPP模块做了升级，在其中添加了BN层。改进后的ASPP如下图所示。

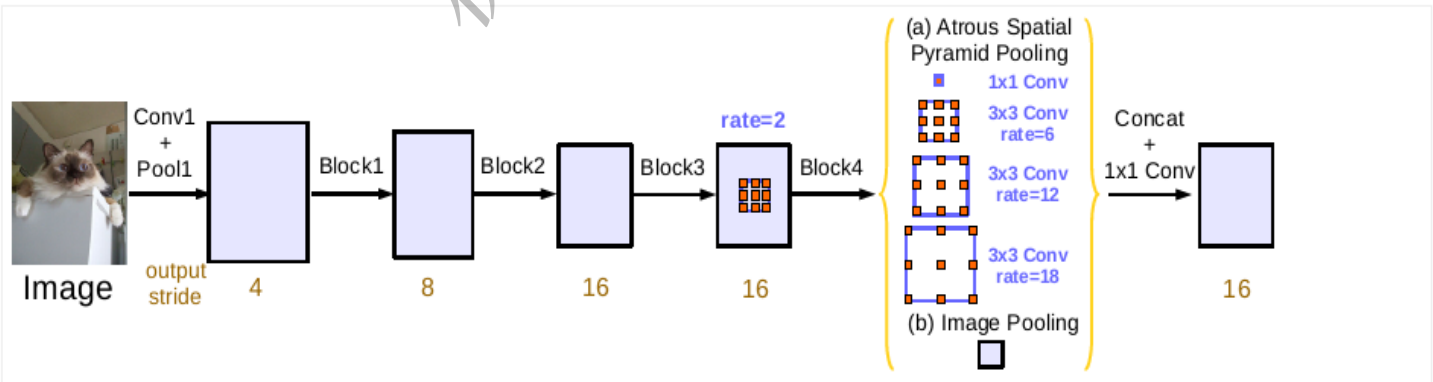
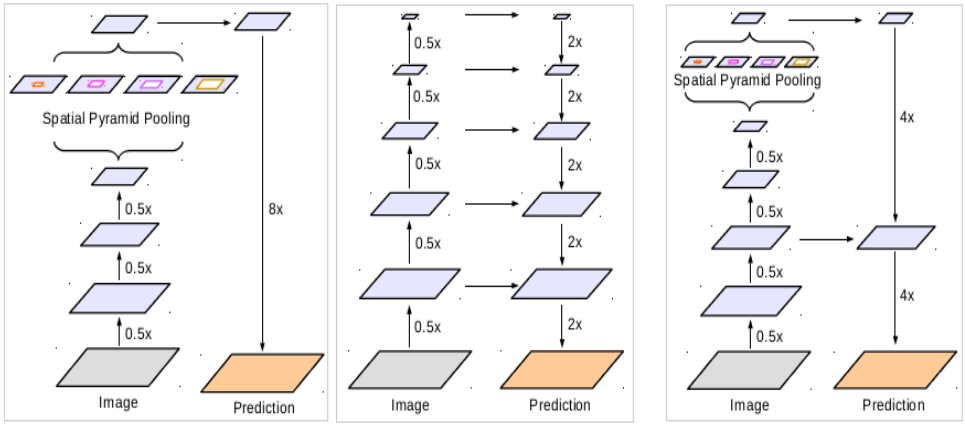


Fig20. ASPP of Deeplab v3

Deeplab v3+在Deeplab v3的基础上做了扩展和改进，其主要改进就是在编解码结构上使用了ASPP。Deeplab v3+可以视作是融合了语义分割两大流派的一项工作，即编解码+ASPP结构。另外Deeplab v3+的Backbone换成了Xception，其深度可分离卷积的设计使得分割网络更加高效。Deeplab v3+结构如下图所示。



(a) Spatial Pyramid Pooling (b) Encoder-Decoder (c) Encoder-Decoder with Atrous Conv

Fig21. ASPP

关于Deeplab系列各个版本的技术点构成总结如下表所示。Deeplab系列算法实现可参考GitHub上各版本，这里不再一一给出。

	Deeplab v1	Deeplab v2	Deeplab v3	Deeplab v3+
Backbone	VGG16	ResNet	ResNet+	Xception
Dilated Conv	✓	✓	✓	✓
ASPP	×	ASPP	ASPP+	ASPP+
CRF	✓	✓	×	×
Encoder-Decoder	×	×	×	✓

Fig22. Summary of Deeplab series.

4.5 PSPNet

PSPNet是针对多尺度问题提出的另一种代表性分割网络。PSPNet认为此前的分割网络没有引入足够的上下文信息及不同感受野下的全局信息而存在分割出现错误的情况，因而引入Global-Scence-Level的信息解决该问题，其Backbone网络也是ResNet。简单来说，PSPNet就是将Deeplab的ASPP模块之前的特征图Pooling了四种尺度，然后将原始特征图和四种Pooling之后的特征图进行合并到一起，再经过一系列卷积之后进行预测的过程。PSPNet结构如下图所示。

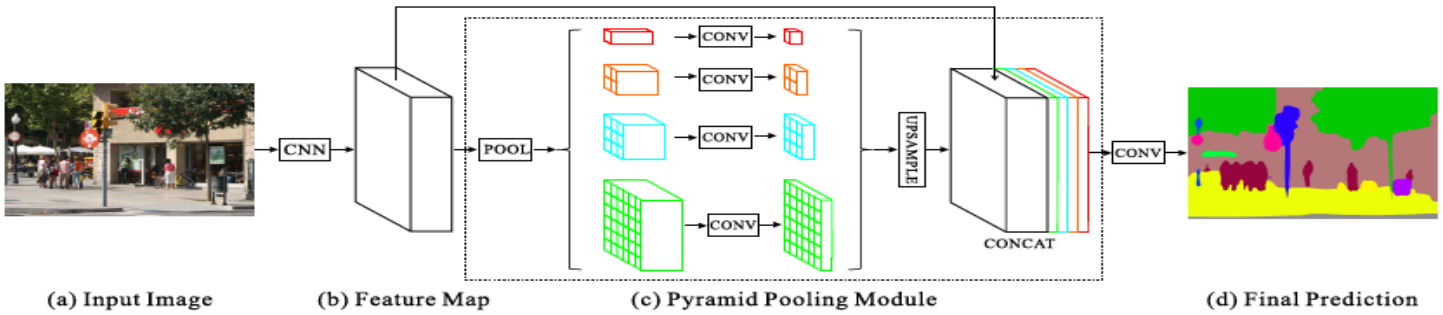


Fig23. PSPNet

一个简易的带有深监督的PSPNet PPM模块写法如下：

```
# Pyramid Pooling Module
class PPMDeepsup(nn.Module):
    def __init__(self, num_class=150, fc_dim=4096,
                 use_softmax=False, pool_scales=(1, 2, 3, 6)):
        super(PPMDeepsup, self).__init__()
        self.use_softmax = use_softmax
        # PPM
        self.ppm = []
        for scale in pool_scales:
            self.ppm.append(nn.Sequential(
                nn.AdaptiveAvgPool2d(scale),
                nn.Conv2d(fc_dim, 512, kernel_size=1, bias=False),
                BatchNorm2d(512),
                nn.ReLU(inplace=True)
            ))
        self.ppm = nn.ModuleList(self.ppm)
        # Deep Supervision
        self.cbr_deepsup = conv3x3_bn_relu(fc_dim // 2, fc_dim // 4, 1)

        self.conv_last = nn.Sequential(
            nn.Conv2d(fc_dim+len(pool_scales)*512, 512,
                      kernel_size=3, padding=1, bias=False),
            BatchNorm2d(512),
            nn.ReLU(inplace=True),
            nn.Dropout2d(0.1),
            nn.Conv2d(512, num_class, kernel_size=1)
        )
        self.conv_last_deepsup = nn.Conv2d(fc_dim // 4, num_class, 1, 1, 0)
        self.dropout_deepsup = nn.Dropout2d(0.1)
```


4.6 UNet++

自从2015年UNet网络提出后，这么多年大家没少在这个U形结构上折腾。大部分做语义分割的朋友都没少在UNet结构上做各种魔改，如果把UNet++算作是UNet的一种魔改的话，那它一定是最成功的魔改

者。

UNet++是一种嵌套的U-Net结构，即内置了不同深度的UNet网络，并且利用了全尺度的跳跃连接（skip connection）和深度监督（deep supervisions）。另外UNet++还设计一种剪枝方案，加快了UNet++的推理速度。UNet++的结构示意图如下所示。

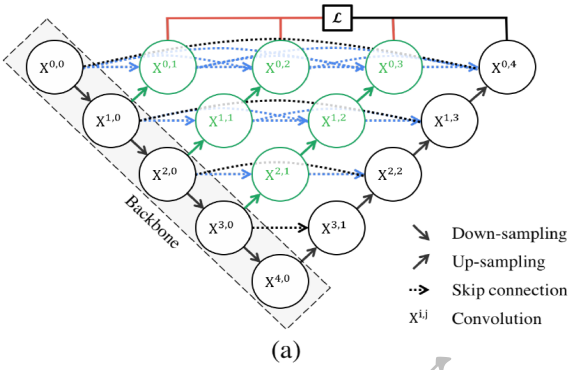


Fig24. UNet++

单纯从结构设计的角度来看，UNet++效果好要归功于其嵌套结构和重新设计的跳跃连接，旨在解决UNet的两个关键挑战:1）优化整体结构的未知深度和2）跳跃连接的不必要的限制性设计。UNet++的一个简单的实现代码如下所示。

```python
import torch
from torch import nn

class NestedUNet(nn.Module):
    def __init__(self, num_classes, input_channels=3, deep_supervision=False, **kwargs):
        super().__init__()
        nb_filter = [32, 64, 128, 256, 512]
        self.deep_supervision = deep_supervision

        self.pool = nn.MaxPool2d(2, 2)
        self.up = nn.Upsample(scale_factor=2, mode='bilinear', align_corners=True)

        self.conv0_0 = VGGBlock(input_channels, nb_filter[0], nb_filter[0])
        self.conv1_0 = VGGBlock(nb_filter[0], nb_filter[1], nb_filter[1])
        self.conv2_0 = VGGBlock(nb_filter[1], nb_filter[2], nb_filter[2])
        self.conv3_0 = VGGBlock(nb_filter[2], nb_filter[3], nb_filter[3])
        self.conv4_0 = VGGBlock(nb_filter[3], nb_filter[4], nb_filter[4])

        self.conv0_1 = VGGBlock(nb_filter[0]+nb_filter[1], nb_filter[0], nb_filter[0])
        self.conv1_1 = VGGBlock(nb_filter[1]+nb_filter[2], nb_filter[1], nb_filter[1])
        self.conv2_1 = VGGBlock(nb_filter[2]+nb_filter[3], nb_filter[2], nb_filter[2])
        self.conv3_1 = VGGBlock(nb_filter[3]+nb_filter[4], nb_filter[3], nb_filter[3])

        self.conv0_2 = VGGBlock(nb_filter[0]*2+nb_filter[1], nb_filter[0], nb_filter[0])
        self.conv1_2 = VGGBlock(nb_filter[1]*2+nb_filter[2], nb_filter[1], nb_filter[1])
        self.conv2_2 = VGGBlock(nb_filter[2]*2+nb_filter[3], nb_filter[2], nb_filter[2])

        self.conv0_3 = VGGBlock(nb_filter[0]*3+nb_filter[1], nb_filter[0], nb_filter[0])
        self.conv1_3 = VGGBlock(nb_filter[1]*3+nb_filter[2], nb_filter[1], nb_filter[1])

        self.conv0_4 = VGGBlock(nb_filter[0]*4+nb_filter[1], nb_filter[0], nb_filter[0])

        if self.deep_supervision:
            self.final1 = nn.Conv2d(nb_filter[0], num_classes, kernel_size=1)
            self.final2 = nn.Conv2d(nb_filter[0], num_classes, kernel_size=1)
            self.final3 = nn.Conv2d(nb_filter[0], num_classes, kernel_size=1)
            self.final4 = nn.Conv2d(nb_filter[0], num_classes, kernel_size=1)
        else:
            self.final = nn.Conv2d(nb_filter[0], num_classes, kernel_size=1)


    def forward(self, input):
        x0_0 = self.conv0_0(input)
        x1_0 = self.conv1_0(self.pool(x0_0))
        x0_1 = self.conv0_1(torch.cat([x0_0, self.up(x1_0)], 1))

        x2_0 = self.conv2_0(self.pool(x1_0))
        x1_1 = self.conv1_1(torch.cat([x1_0, self.up(x2_0)], 1))
        x0_2 = self.conv0_2(torch.cat([x0_0, x0_1, self.up(x1_1)], 1))

        x3_0 = self.conv3_0(self.pool(x2_0))
```

```
        x2_1 = self.conv2_1(torch.cat([x2_0, self.up(x3_0)], 1))
        x1_2 = self.conv1_2(torch.cat([x1_0, x1_1, self.up(x2_1)], 1))
        x0_3 = self.conv0_3(torch.cat([x0_0, x0_1, x0_2, self.up(x1_2)], 1))

        x4_0 = self.conv4_0(self.pool(x3_0))
        x3_1 = self.conv3_1(torch.cat([x3_0, self.up(x4_0)], 1))
        x2_2 = self.conv2_2(torch.cat([x2_0, x2_1, self.up(x3_1)], 1))
        x1_3 = self.conv1_3(torch.cat([x1_0, x1_1, x1_2, self.up(x2_2)], 1))
        x0_4 = self.conv0_4(torch.cat([x0_0, x0_1, x0_2, x0_3, self.up(x1_3)], 1))

        if self.deep_supervision:
            output1 = self.final1(x0_1)
            output2 = self.final2(x0_2)
            output3 = self.final3(x0_3)
            output4 = self.final4(x0_4)
            return [output1, output2, output3, output4]
        else:
            output = self.final(x0_4)
            return output
```

完整实现过程可参考GitHub开源代码。

以上仅对几个主要的语义分割网络模型进行介绍，从当年的FCN到如今的各种模型层出不穷，想要对所有的SOTA模型全部进行介绍已经不太可能。其他诸如ENet、DeconvNet、RefineNet、HRNet、PixelNet、BiSeNet、UpperNet等网络模型，均各有千秋。本小节旨在让大家熟悉语义分割的主要模型结构和设计。深度学习和计算机视觉发展日新月异，一个新的SOTA模型出来，肯定很快就会被更新的结构设计所代替，重点是我们要了解语义分割的发展脉络，对主流的前沿研究能够保持一定的关注。

5. 语义分割训练Tips

PyTorch是一款极为便利的深度学习框架。在日常实验过程中，我们要多积累和总结，假以时日，人人都能总结出一套自己的高效模型搭建和训练套路。这一节我们给出一些惯用的PyTorch代码搭建方式，以及语义分割训练过程中的可视化方法，方便大家在训练过程中能够直观的看到训练效果。

5.1 PyTorch代码搭建方式

无论是分类、检测还是分割抑或是其他非视觉的深度学习任务，其代码套路相对来说较为固定，不会跳出基本的代码框架。一个深度学习的实现代码框架无非就是以下五个主要构成部分：

- 数据：Data
- 模型：Model
- 判断：Criterion
- 优化：Optimizer
- 日志：Logger

所以一个基本的顺序实现范式如下：

```
# data
dataset = VOC()||COCO()||ADE20K()
data_loader = data.DataLoader(dataSet)

# model
model = ...
model_parallel = torch.nn.DataParallel(model)

# Criterion
loss = criterion(...)

# Optimizer
optimer = optim.SGD(...)

# Logger and Visulization
visdom = ...
tensorboard = ...
textlog = ...

# Model Parameters
data_size, batch_size, epoch_size, iterations = ..., ...
```

不论是哪种深度学习任务，一般都免不了以上五项基本模块。所以一个简单的、相对完整的PyTorch模型项目代码应该是如下结构的：

```
|-- semantic segmentation example
    |-- dataset.py
    |-- models
        |-- unet.py
        |-- deeplabv3.py
        |-- pspnet.py
        |-- ...
    |-- _config.yml
    |-- main.py
    |-- utils
    |   |-- visual.py
    |   |-- loss.py
    |   |-- ...
    |-- README.md
    ...
```

上面的示例代码结构中，我们把训练和验证相关代码都放到 `main.py` 文件中，但在实际实验中，这块的灵活性极大。一般来说，模型训练策略有三种，一种是边训练边验证最后再测试、另一种则是在训练中验证，将验证过程糅合到训练过程中，还有一种最简单，就是训练完了再单独验证和测试。所以，我们这里也可以单独定义对应的函数，训练 `train()` 、验证 `val()` 以及测试 `test()` 除此之外，还有一些辅助

功能需要设计，包括打印训练信息 `print()` 、绘制损失函数 `plot()` 、保存最优模型 `save()` ， 调整训练参数 `update()` 。

所以训练代码控制流程可以归纳为TVT+PPSU的模式。

5.2 可视化方法

PyTorch原生的可视化支持模块是Visdom，当然鉴于TensorFlow的应用广泛性，PyTorch同时也支持TensorBoard的可视化方法。语义分割需要能够直观的看到训练效果，所以在训练过程中辅以一定的可视化方法是十分必要的。

Visdom

visdom是一款用于创建、组织和共享实时大量训练数据可视化的灵活工具。深度学习模型训练通常放在远程的服务器上，服务器上训练的一个问题就在于不能方便地对训练进行可视化，相较于TensorFlow的可视化工具TensorBoard，visdom则是对应于PyTorch的可视化工具。直接通过 `pip install visdom` 即可完成安装，之后在终端输入如下命令即可启动visdom服务：

```
python -m visdom.server
```

启动服务后输入本地或者远程地址，端口号8097，即可打开visdom主页。具体到深度学习训练时，我们可以在torch训练代码下插入visdom的可视化模块：

```
if args.steps_plot > 0 and step % args.steps_plot == 0:
    image = inputs[0].cpu().data
    vis.image(image,f'input (epoch: {epoch}, step: {step})')
    vis.image(outputs[0].cpu().max(0)[1].data, f'output (epoch: {epoch}, step: {step})')
    vis.image(targets[0].cpu().data, f'target (epoch: {epoch}, step: {step})')
    vis.image(loss, f'loss (epoch: {epoch}, step: {step})')
```

visdom效果展示如下：

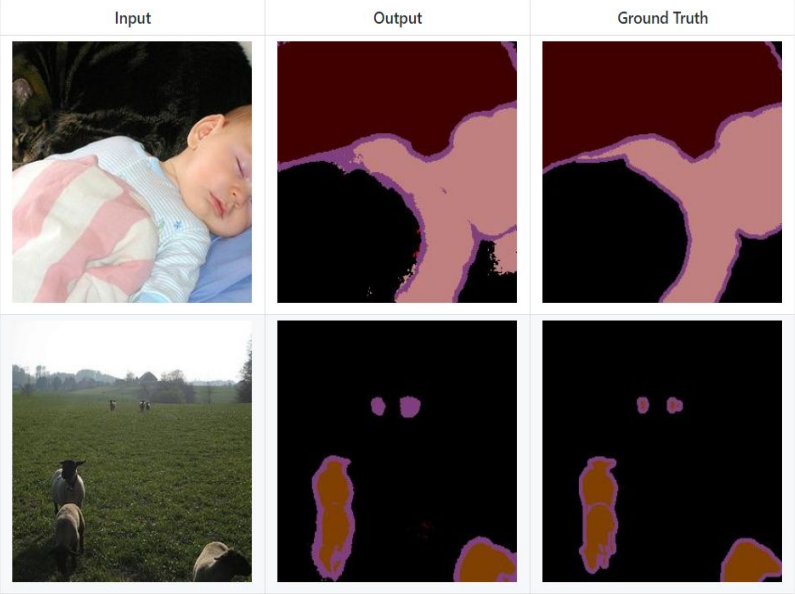


Fig25. visdom example

TensorBoard

很多TensorFlow用户更习惯于使用TensorBoard来进行训练的可视化展示。为了能让PyTorch用户也能用上TensorBoard，有开发者提供了PyTorch版本的TensorBoard，也就是tensorboardX。熟悉TensorBoard的用户可以无缝对接到tensorboardX，安装方式为：

```
pip install tensorboardX
```

除了要安装PyTorch之外，还需要安装TensorFlow。跟TensorBoard一样，tensorboardX也支持scalar, image, figure, histogram, audio, text, graph, onnx_graph, embedding, pr_curve，video等不同类型对象的可视化展示方式。tensorboardX和TensorBoard的启动方式一样，直接在终端下运行：

```
tensorboard --logdir runs
```

一个完整tensorboardX使用demo如下：

```
import torch
import torchvision.utils as vutils
import numpy as np
import torchvision.models as models
from torchvision import datasets
from tensorboardX import SummaryWriter

resnet18 = models.resnet18(False)
writer = SummaryWriter()
sample_rate = 44100
freqs = [262, 294, 330, 349, 392, 440, 440, 440, 440, 440, 440]

for n_iter in range(100):

    dummy_s1 = torch.rand(1)
    dummy_s2 = torch.rand(1)
    # data grouping by `slash`
    writer.add_scalar('data/scalar1', dummy_s1[0], n_iter)
    writer.add_scalar('data/scalar2', dummy_s2[0], n_iter)

    writer.add_scalars('data/scalar_group', {'xsinx': n_iter * np.sin(n_iter),
                                             'xcosx': n_iter * np.cos(n_iter),
                                             'arctanx': np.arctan(n_iter)}, n_iter)

    dummy_img = torch.rand(32, 3, 64, 64)  # output from network
    if n_iter % 10 == 0:
        x = vutils.make_grid(dummy_img, normalize=True, scale_each=True)
        writer.add_image('Image', x, n_iter)

        dummy_audio = torch.zeros(sample_rate * 2)
        for i in range(x.size(0)):
            # amplitude of sound should in [-1, 1]
            dummy_audio[i] = np.cos(freqs[n_iter // 10] * np.pi * float(i) / float(sample_rate))
        writer.add_audio('myAudio', dummy_audio, n_iter, sample_rate=sample_rate)

        writer.add_text('Text', 'text logged at step:' + str(n_iter), n_iter)

        for name, param in resnet18.named_parameters():
            writer.add_histogram(name, param.clone().cpu().data.numpy(), n_iter)

        # needs tensorboard 0.4RC or later
        writer.add_pr_curve('xoxo', np.random.randint(2, size=100), np.random.rand(100), n_iter)

dataset = datasets.MNIST('mnist', train=False, download=True)
images = dataset.test_data[:100].float()
label = dataset.test_labels[:100]

features = images.view(100, 784)
writer.add_embedding(features, metadata=label, label_img=images.unsqueeze(1))

# export scalar data to JSON for external processing
```

```
writer.export_scalars_to_json("./all_scalars.json")
writer.close()
```

tensorboardX的展示界面如图所示。

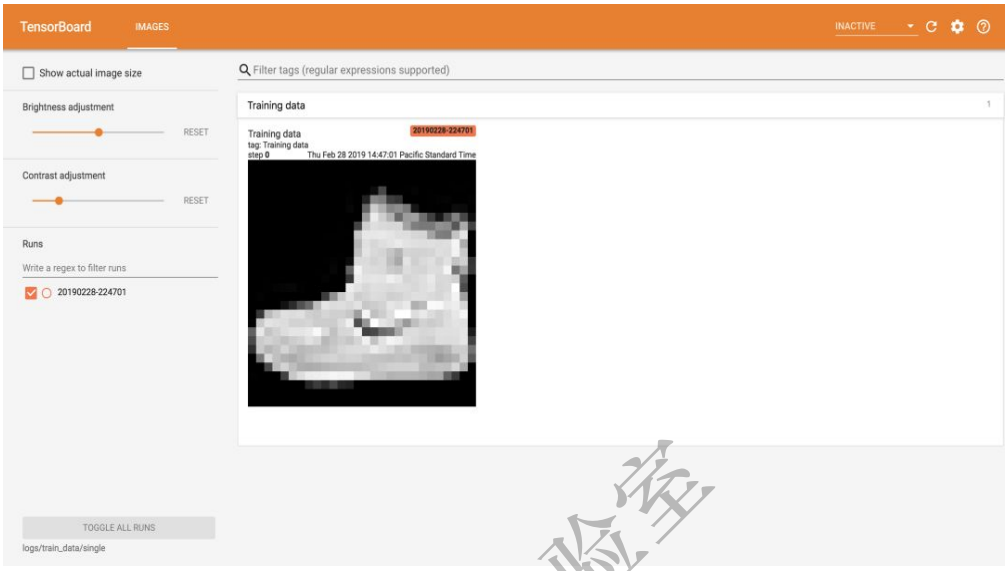


Fig26. tensorboardX example

参考文献

1. awesome-semantic-segmentation
2. Long J , Shelhamer E , Darrell T . Fully Convolutional Networks for Semantic Segmentation[J]. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2015, 39(4):640-651.
3. Ronneberger O , Fischer P , Brox T . U-Net: Convolutional Networks for Biomedical Image Segmentation[J]. 2015.
4. Badrinarayanan V , Kendall A , Cipolla R . SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation[J]. 2015.
5. Zhao H , Shi J , Qi X , et al. Pyramid Scene Parsing Network[J]. 2016.
6. Huang H , Lin L , Tong R , et al. UNet 3+: A Full-Scale Connected UNet for Medical Image Segmentation[J]. arXiv, 2020.
7. UNet++: A Nested U-Net Architecture for Medical Image Segmentation
8. https://github.com/4uiiurz1/pytorch-nested-unet
9. Minaee S , Boykov Y , Porikli F , et al. Image Segmentation Using Deep Learning: A Survey[J]. 2020.
10. Guo Y , Liu Y , Georgiou T , et al. A review of semantic segmentation using deep neural networks[J]. International Journal of Multimedia Information Retrieval, 2017.