


# U-Net的代码实现与相关细节

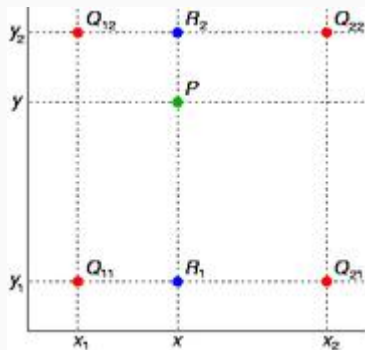
Harry  
2019.12.7



# 上次课程相关问题

**k=2n, s=n?**

VGG中卷积核的初始化可以使用pre-trained ImageNet, 转置卷积初始化如果使用随机初始化的权重, 将会花费更多的时间, 因此需要使用一种更合理有效的方法。这里就要用到 bilinear kernel。



$$f(x, y) \approx \frac{f(Q_{11})}{(x_2 - x_1)(y_2 - y_1)}(x_2 - x)(y_2 - y) + \frac{f(Q_{21})}{(x_2 - x_1)(y_2 - y_1)}(x - x_1)(y_2 - y) \\ + \frac{f(Q_{12})}{(x_2 - x_1)(y_2 - y_1)}(x_2 - x)(y - y_1) + \frac{f(Q_{22})}{(x_2 - x_1)(y_2 - y_1)}(x - x_1)(y - y_1).$$

# 上次课程相关问题

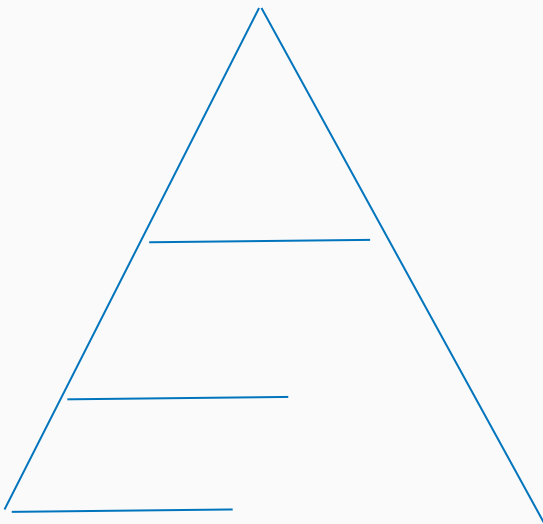
## $k=2n$ , $s=n$ ?

VGG中卷积核的初始化可以使用pre-trained ImageNet, 转置卷积初始化如果使用随机初始化的权重, 将会花费更多的时间, 因此需要使用一种更合理有效的方法。这里就要用到 bilinear kernel。

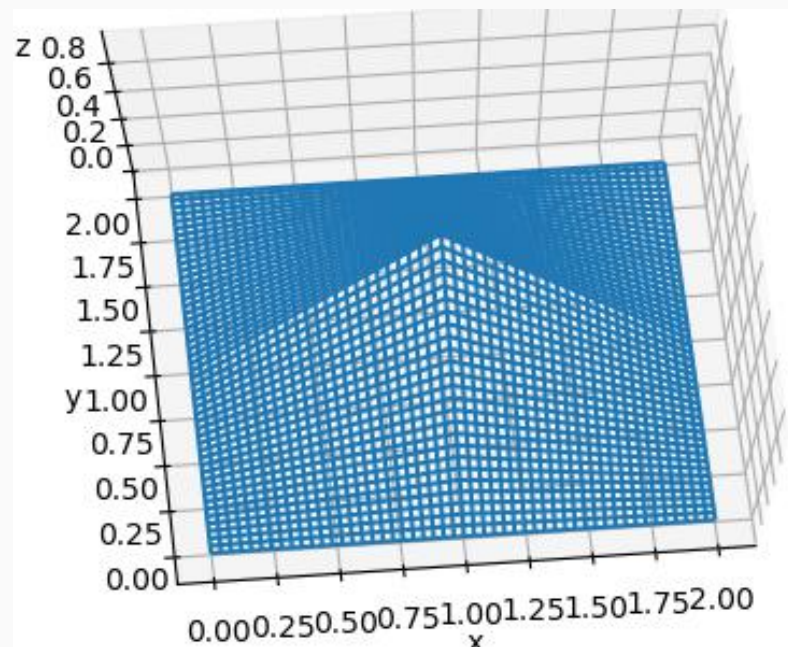
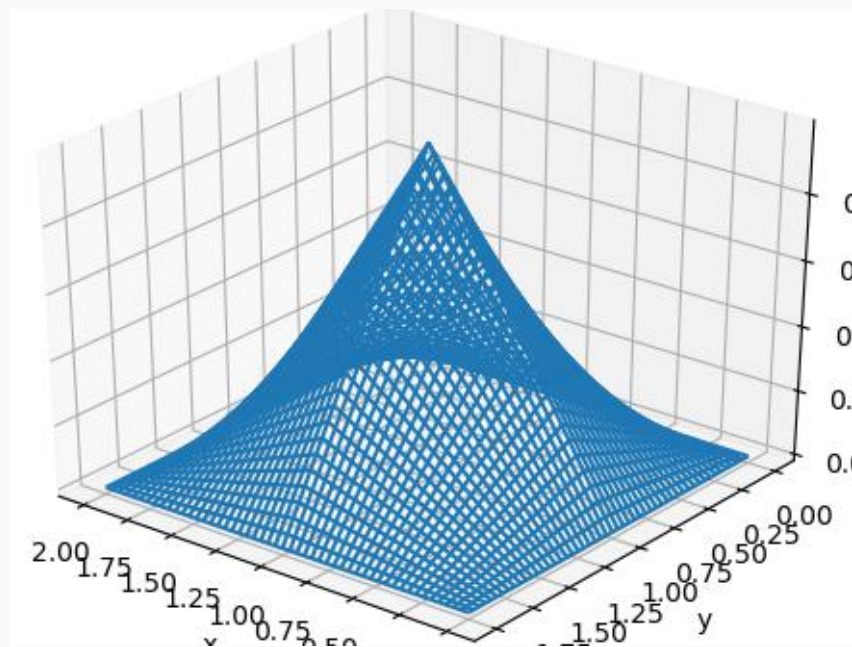
```
def bilinear_kernel(in_channels, out_channels, kernel_size):  
    """  
    return a bilinear filter tensor  
    """  
  
    factor = (kernel_size + 1) // 2  
    if kernel_size % 2 == 1:  
        center = factor - 1  
    else:  
        center = factor - 0.5  
    og = np.ogrid[:kernel_size, :kernel_size]  
    filt = (1 - abs(og[0] - center) / factor) * (1 - abs(og[1] - center) / factor)  
    weight = np.zeros((in_channels, out_channels, kernel_size, kernel_size), dtype=np.float32)  
    weight[range(in_channels), range(out_channels), :, :] = filt  
    return torch.from_numpy(weight)
```

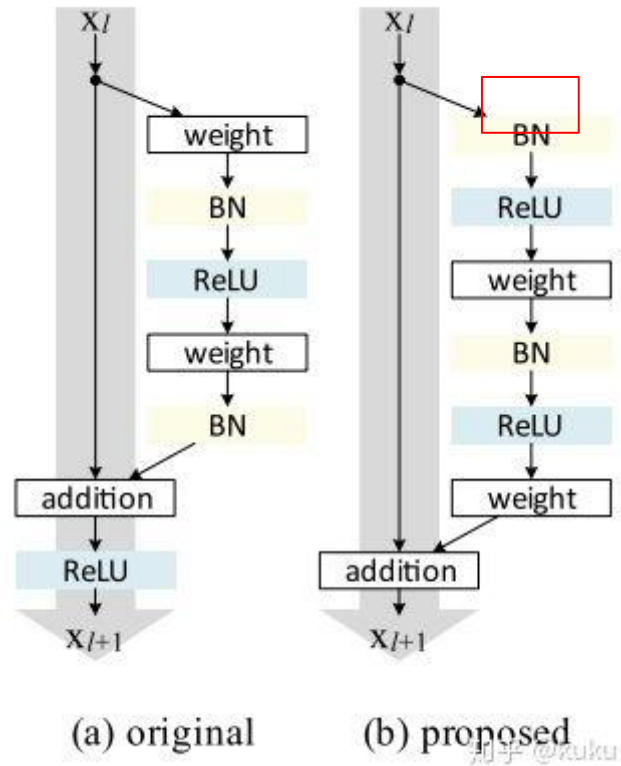
## 上次课程相关问题

$$\begin{aligned}& (1 - | \frac{x-c}{f} |) (1 - | \frac{y-c}{f} |) \\& + (1 - | \frac{x+s-c}{f} |) (1 - | \frac{y-c}{f} |) \\& + (1 - | \frac{x-c}{f} |) (1 - | \frac{y+s-c}{f} |) \\& + (1 - | \frac{x+s-c}{f} |) (1 - | \frac{y+s-c}{f} |) \\& = (1 - \frac{c-x}{f}) (1 - \frac{c-y}{f}) + (1 - \frac{x+s-c}{f}) (1 - \frac{c-y}{f}) \\& \quad (1 - \frac{c-x}{f}) (1 - \frac{y+s-c}{f}) + (1 - \frac{x+s-c}{f}) (1 - \frac{y+s-c}{f}) \\& = (1 - \frac{c-y}{f}) (2 - \frac{s}{f}) + \\& \quad (1 - \frac{y+s-c}{f}) (2 - \frac{s}{f}) \\& = (2 - \frac{s}{f}) (2 - \frac{s}{f}) = 1 \\& \Downarrow \\& 2 - \frac{s}{f} = 1 \Rightarrow s = f \quad \therefore\end{aligned}$$



## 上次课程相关问题





增加了skip-connected好处:

1. 信息传递与优化更加有效
2. 能够组成不同深度的网络
3. 每层的特征也可以看成融合了多个层级的特征(分布式特征)

# Batch Normalization

**Input:** Values of  $x$  over a mini-batch:  $\mathcal{B} = \{x_{1\dots m}\}$ ;

Parameters to be learned:  $\gamma, \beta$

**Output:**  $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{ mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{ mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{ normalize}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{ scale and shift}$$

操作过程为：先求出每个channel的均值与方差，

然后对每个通道的数据进行归一化。

维度变化为：NHWC---->C

注意：在训练时的均值与方差是当前批次的均值与方差，测试时使用的是移动平均值。

## Batch Normalization

优点:

- 训练速度更快。因为网络的数据分布更加稳定，模型更容易学习。
- 使用更大的学习率。因为网络的数据分布更加稳定，使用更大的学习率不会轻易造成损失函数曲线发散情况。
- 不需要太关注模型参数的初始化。模型的随机初始化结果对模型的训练没有太大的影响。
- 正则化效果。Mini-batch 的 BN 层是使用 mini-batch 的统计值近似训练集的统计值，使得 BN 层具有正则化效果。





1. 原文中作者是将bn放在激活函数之前。
2. [知乎上有个高赞回答](#): 放在relu后面比较好。

## BN -- before or after ReLU?

Name	Accuracy	LogLoss	Comments
Before	0.474	2.35	As in paper
Before + scale&bias layer	0.478	2.33	As in paper
After	<b>0.499</b>	<b>2.21</b>	
After + scale&bias layer	0.493	2.24	



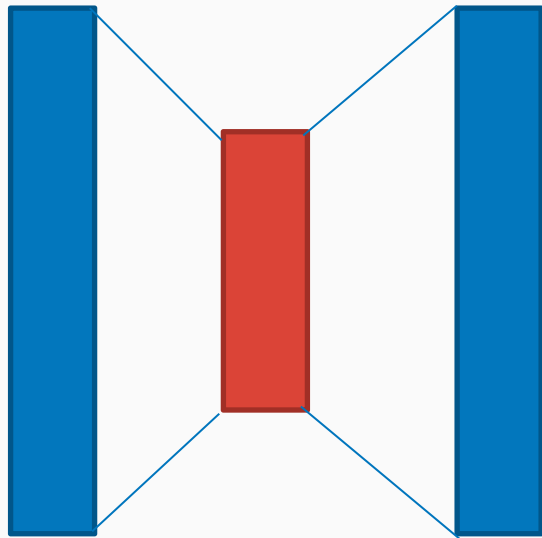
1. resnet: bn放在relu之前;
2. densenet: bn放在relu之前;
3. mobilenet: bn放在relu之前

# Bottleneck Architectures

$$\begin{bmatrix} 1 \times 1, 64 \\ 3 \times 3, 64 \\ 1 \times 1, 256 \end{bmatrix}$$

Bottleneck的作用:

1. 减少了参数与计算
2. 具有更好的特征提取能力

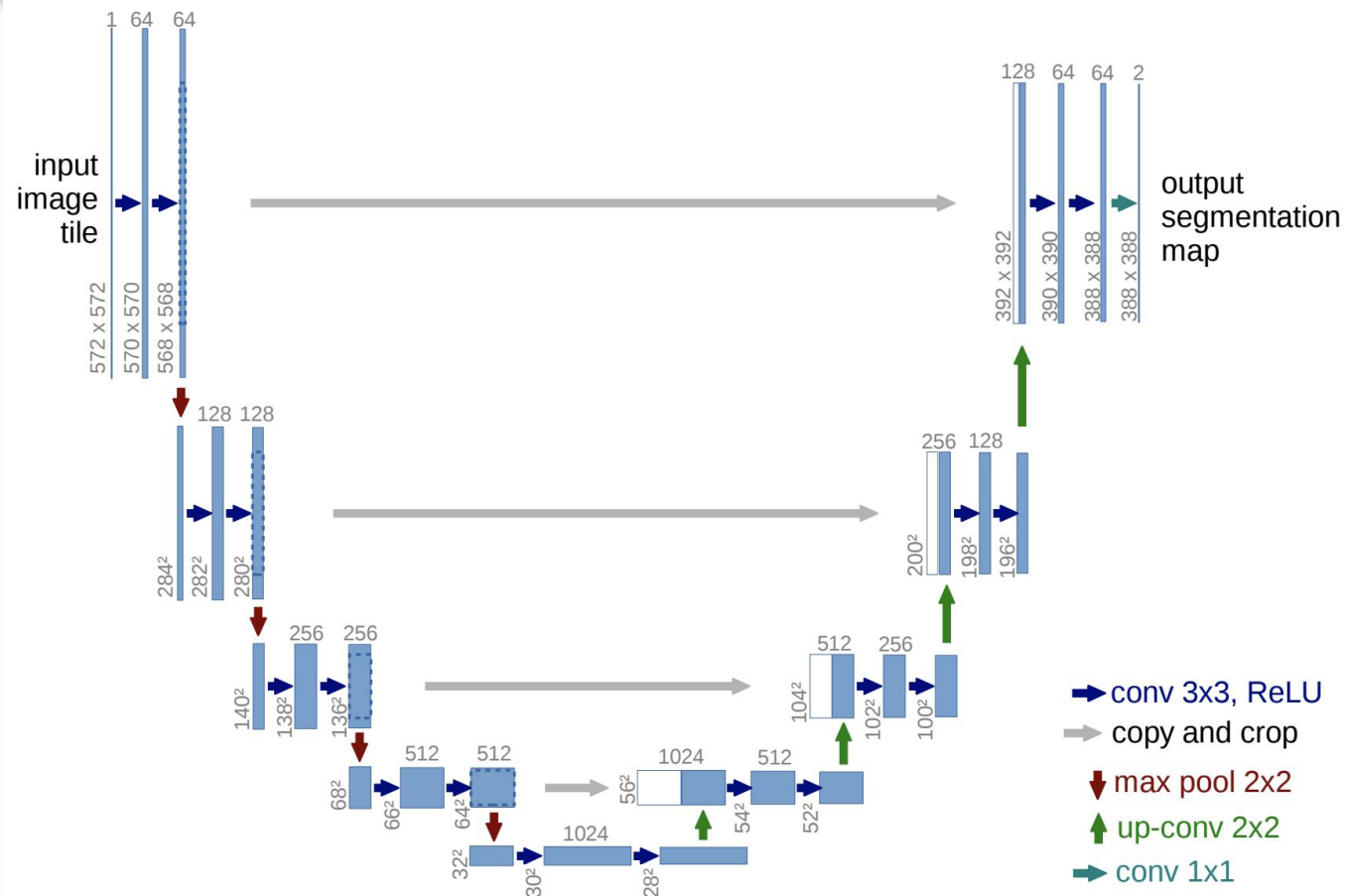


# ResNet101

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix} \times 3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix} \times 4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix} \times 8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix} \times 6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix} \times 6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix} \times 23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix} \times 36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix} \times 3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		$1.8 \times 10^9$	$3.6 \times 10^9$	$3.8 \times 10^9$	$7.6 \times 10^9$	$11.3 \times 10^9$

ures for ImageNet. Building blocks are shown in brackets (see also Fig. 5), with the numbers of block

# UNet尺寸变化与细节



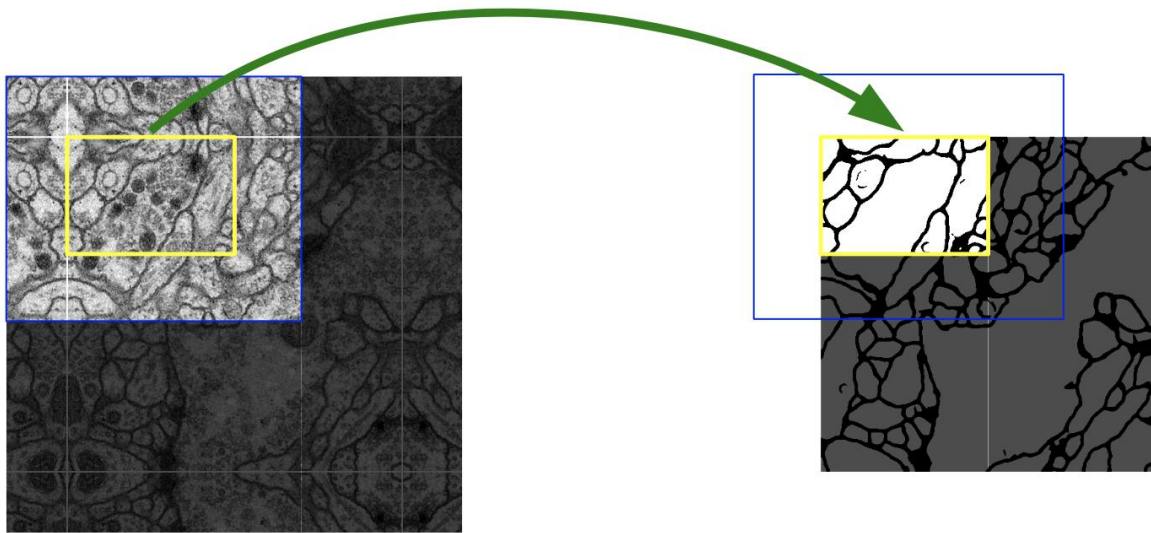
1. 第一个细节上采样通道数是当前的一半。

2. 这里是裁剪加concat的操作。

3. 原始作者的实现中unet实际上网络比较浅。

4. padding都为0

## UNet中的其他细节



$$w(\mathbf{x}) = w_c(\mathbf{x}) + w_0 \cdot \exp \left( -\frac{(d_1(\mathbf{x}) + d_2(\mathbf{x}))^2}{2\sigma^2} \right)$$

## 我们的实现

1. 使用**encoder-decoder**的编码范式，便于模型扩展与修改。
2. 我们上采样到原始图像的**1/4**的倍数。
3. 思考**add**与**concat**操作的异同。