Lesson 14

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

目录 Contents

目录 Contents

1.1 简述

- 低分辨率图像（72*72）
- 转换为高分辨率图像（288*288）
- 细节部分需要补全

1.2 数据

- 与之前大致相同，但是标签全部为0；
- 从ImageNet数据集中随机选择2%数据。

In [32]:
```
fnames_full,label_arr_full,all_labels = folder_source(PATH, 'train')
fnames_full = ['/'.join(Path(fn).parts[-2:]) for fn in fnames_full]
list(zip(fnames_full[:5],label_arr_full[:5]))
```

Out[32]:
```
[('n01440764/n01440764_11787.JPEG', 0),
 ('n01440764/n01440764_12732.JPEG', 0),
 ('n01440764/n01440764_4934.JPEG', 0),
 ('n01440764/n01440764_8063.JPEG', 0),
 ('n01440764/n01440764_26631.JPEG', 0)]
```

In [33]:
```
all_labels[:5]
```

Out[33]:
```
['n01440764', 'n01443537', 'n01491361', 'n01494475', 'n01498041']
```

In [34]:
```
np.random.seed(42)
keep_pct = 1.
# keep_pct = 0.02
keeps = np.random.rand(len(fnames_full)) < keep_pct
fnames = np.array(fnames_full, copy=False)[keeps]
label_arr = np.array(label_arr_full, copy=False)[keeps]
```

1.2 数据

- 数据集继承自图片数据集；

,旋转90度

喿作同时

匕为可显

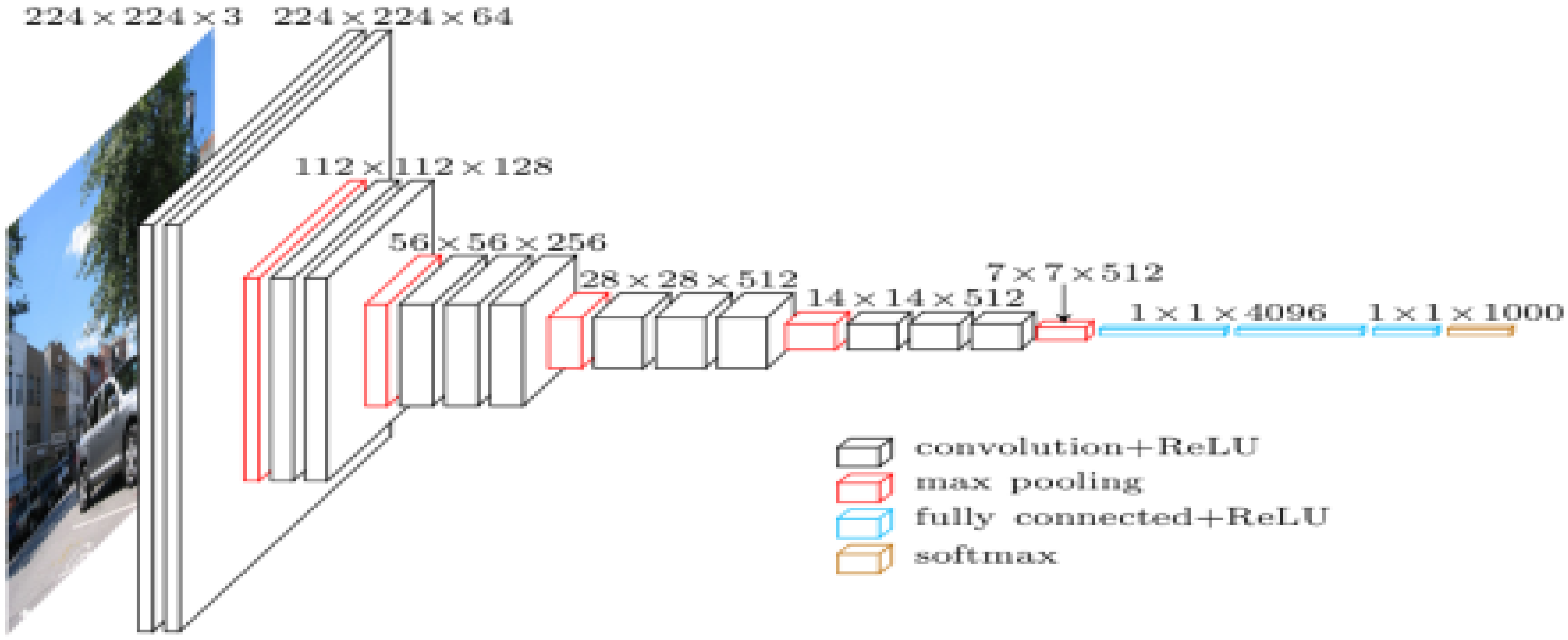
```
In [36]: scale,bs = 2,64
```

```
In [45]: idx=1
         fig, axes = plt.subplots(1, 2, figsize=(9,5))
         show_img(x,idx, ax=axes[0])
         show_img(y,idx, ax=axes[1])
```

```
         ax.imshow(np.clip(ims,0,1)[idx])
         ax.axis('off')
```

```
In [44]: x,y = next(iter(md.val_dl))
         x.size(),y.size()
Out[44]: (torch.Size([64, 3, 72, 72]), torch.Size([64, 3, 144, 144]))
```

1.3 VGG16


224 × 224 × 3
224 × 224 × 64
112 × 112 × 128
56 × 56 × 256
28 × 28 × 512
14 × 14 × 512
7 × 7 × 512
1 × 1 × 4096
1 × 1 × 1000
convolution+ReLU
max pooling
fully connected+ReLU
softmax

1.3 VGG16

- 缺点：

  全连接层权重矩阵过大（7*7*512*4096）；

- 优点：

  与许多其他模型（ResNet）相比保留了更多细节，而对超分辨率与图片分割来说细节很重要；

  不使用平均池化保留了空间信息。

1.4 模型

- 目标：小图片->大图片。
- 两种方式：先通过步长为1的层进行运算再进行上采样（upsampling）或是反之，但我们为了减小计算量，选择前种方法。

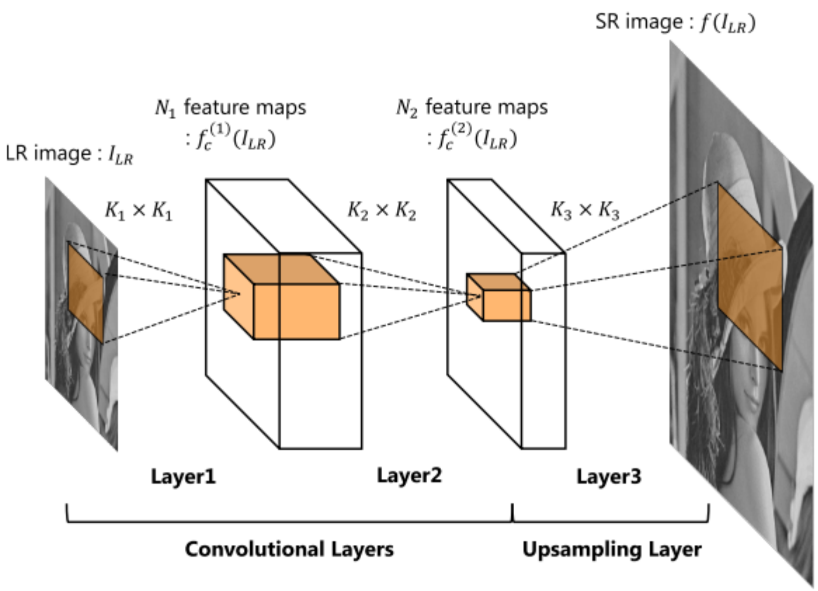
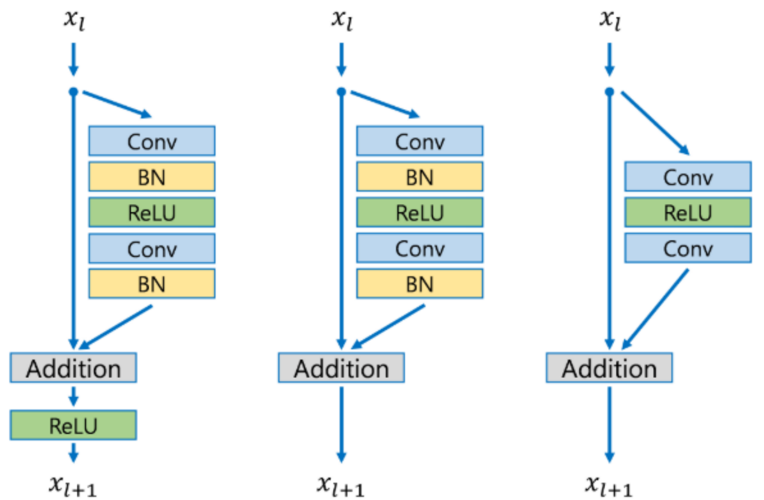


Fig. 2: CNNs with an upsampling Layer

1.5 卷积层

- 不用BatchNorm层的原因：减少内存用量，并且尽可能保留输入的信息，避免标准化导致的取值范围限制，从而影响后续计算。
- 加入scale系数：有助于训练时的稳定性，避免出现activation无限的情况。



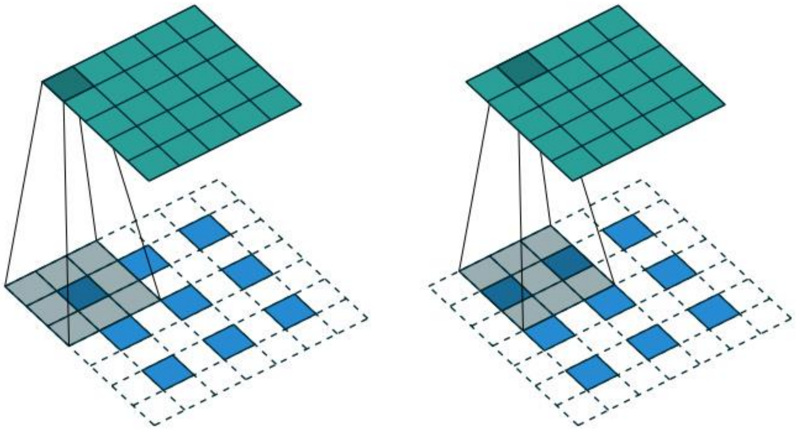
(a) Original (b) SRResNet (c) Proposed

```
In [55]: class ResSequential(nn.Module):
             def __init__(self, layers, res_scale=1.0):
                 super().__init__()
                 self.res_scale = res_scale
                 self.m = nn.Sequential(*layers)

             def forward(self, x): return x + self.m(x) * self.res_scale
```

```
In [56]: def res_block(nf):
             return ResSequential(
                 [conv(nf, nf, actn=True), conv(nf, nf)],
                 0.1)
```

1.6 上采样

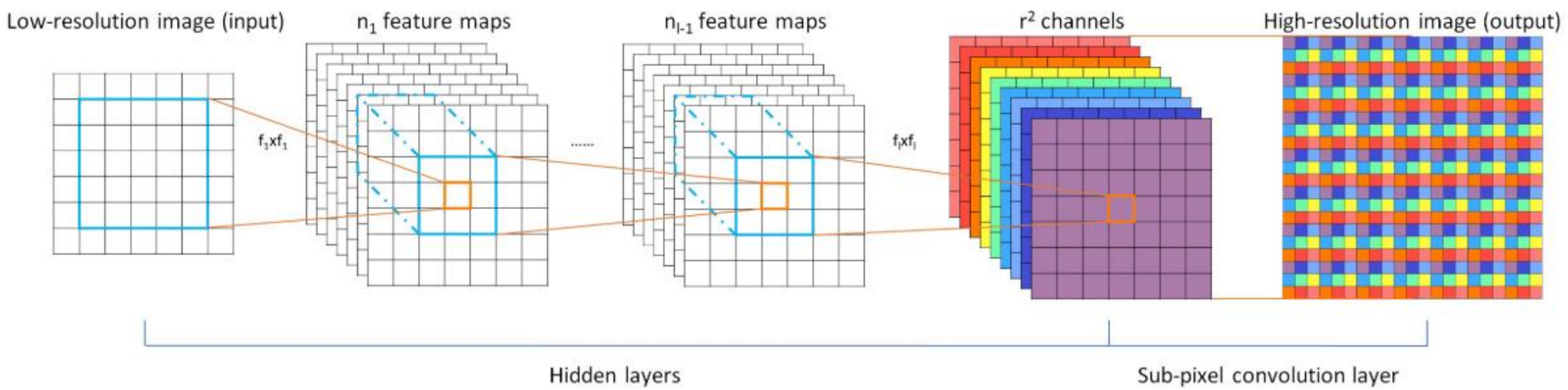
- 反卷积存在的问题：浪费计算量，不同位置的计算方式不同，造成棋盘效应（checkerboard artifacts）。
- 解决方式：像素重组（Pixel Shuffle）。



```
In [57]: def upsample(ni, nf, scale):
             layers = []
             for i in range(int(math.log(scale,2))):
                 layers += [conv(ni, nf*4), nn.PixelShuffle(2)]
             return nn.Sequential(*layers)
```

1.7 像素重组

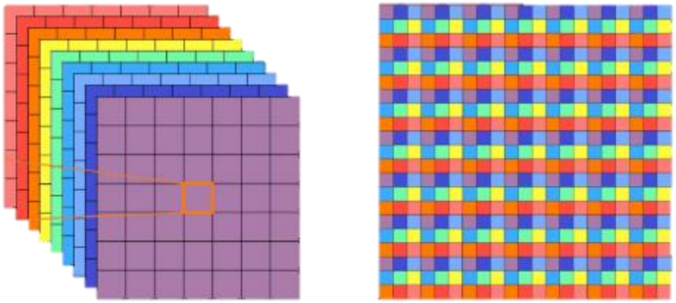
对每一层进行r^2次卷积



根据一定顺序组合

1.7 像素重组

- 结果：仍然存在棋盘效应。



- ICNR:初始化一层并复制到其他层，使亚像素层每个3*3的内容都相同。

将每个subkernel复制scale^{2}次

```
def icnr(x, scale=2, init=nn.init.kaiming_normal):
    new_shape = [int(x.shape[0] / (scale ** 2))] + list(x.shape[1:])
    subkernel = torch.zeros(new_shape)
    subkernel = init(subkernel)
    subkernel = subkernel.transpose(0, 1)
    subkernel = subkernel.contiguous().view(subkernel.shape[0],
                                            subkernel.shape[1], -1)
    kernel = subkernel.repeat(1, 1, scale ** 2)
    transposed_shape = [x.shape[1]] + [x.shape[0]] + list(x.shape[2:])
    kernel = kernel.contiguous().view(transposed_shape)
    kernel = kernel.transpose(0, 1)
    return kernel
```

1.8 像素损失

- 概念：计算生成图像的像素与期望的图像的像素间的均方误差。
- 表现不好的原因：减小均方误差的最好方式就是平均各像素点的值，但是这样会让图像模糊，因此使用感知误差。

1.9 感知误差

- 将真实图片卷积得到的feature与生成图片卷积得到的feature作比较，使得高层信息（内容和全局结构）接近，但是课程中误差的计算同时也考虑了像素误差。

```
class FeatureLoss(nn.Module):
    def __init__(self, m, layer_ids, layer_wgts):
        super().__init__()
        self.m,self.wgts = m,layer_wgts
        self.sfs = [SaveFeatures(m[i]) for i in layer_ids]

    def forward(self, input, target, sum_layers=True):
        self.m(VV(target.data))
        res = [F.l1_loss(input,target)/100]
        targ_feat = [V(o.features.data.clone()) for o in self.sfs]
        self.m(input)
        res += [F.l1_loss(flatten(inp.features),flatten(targ))*wgt
               for inp,targ,wgt in zip(self.sfs, targ_feat, self.wgts)]
        if sum_layers: res = sum(res)
        return res

    def close(self):
        for o in self.sfs: o.remove()
```

1.10 进一步改变大小

```
In [8]:  scale,bs = 2,64
         # scale,bs = 4,32
         sz_hr = sz_lr*scale
```

```
In [20]: def upsample(ni, nf, scale):
             layers = []
             for i in range(int(math.log(scale,2))):
                 layers += [conv(ni, nf*4), nn.PixelShuffle(2)]
             return nn.Sequential(*layers)
```

```
In [32]: learn.freeze_to(999)
```

```
In [33]: for i in range(10,13): set_trainable(m.features[i], True)
```

```
In [34]: conv_shuffle = m.features[10][2][0]
         kernel = icnr(conv_shuffle.weight, scale=scale)
         conv_shuffle.weight.data.copy_(kernel);
```

```
In [31]: m = nn.DataParallel(m, [0,2])
         learn = Learner(md, SingleModel(m), opt_fn=optim.Adam)
```

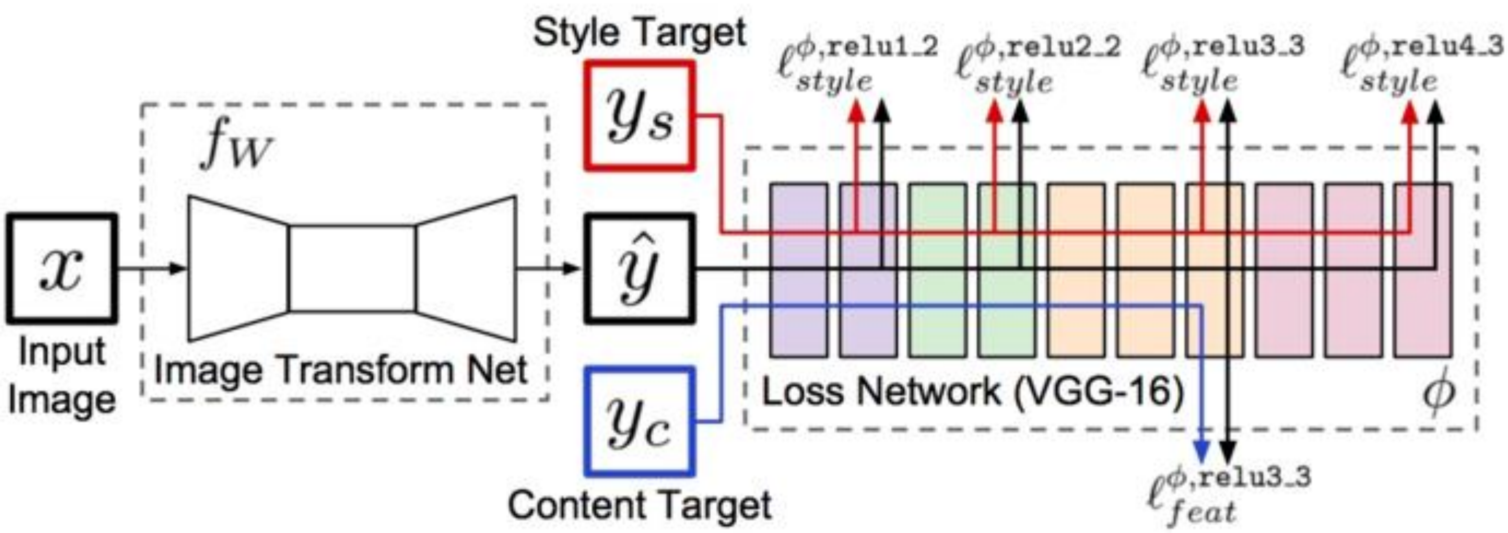
- 重新加载之前保存的模型。
- 问题：upsample函数层数不同。
- 解决方式：在load_state_dict函数调用时将strict设为False，程序自动填充已有的层数，并随机初始化剩下的层数。
- 然后冻结所有层，解冻后面的部分重新进行学习。

1.11 结果

目录 Contents

2.1 风格迁移网络

- 由于超分辨率网络与风格迁移网络的相似性，很容易利用之前已建立的网络转化为风格迁移网络；
- upsample已有，因此需要一个downsample；
- 损失函数需要修改，同时考虑风格图片与原图。



2.2 部分与之前不同的代码

```
def conv(ni, nf, kernel_size=3, stride=1, actn=True, pad=None, bn=True):
    if pad is None: pad = kernel_size//2
    layers = [nn.Conv2d(ni, nf, kernel_size, stride=stride, padding=pad, bias=not bn)]
    if actn: layers.append(nn.ReLU(inplace=True))
    if bn: layers.append(nn.BatchNorm2d(nf))
    return nn.Sequential(*layers)
```

```
class ResSequentialCenter(nn.Module):
    def __init__(self, layers):
        super().__init__()
        self.m = nn.Sequential(*layers)

    def forward(self, x): return x[:, :, 2:-2, 2:-2] + self.m(x)
```

```
def upsample(ni, nf):
    return nn.Sequential(nn.Upsample(scale_factor=2), conv(ni, nf))
```

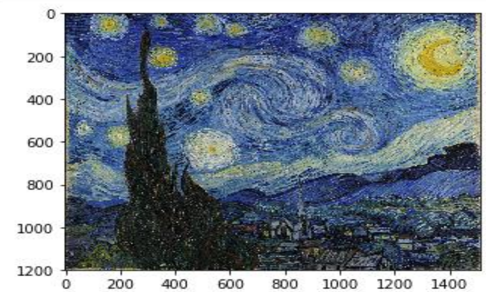
2.3 风格图片的处理

- 由于batch size是24，所以使用broadcast_to来改变数组
- 示例：
- >>> x = np.array([1, 2, 3])
- >>> np.broadcast_to(x, (3, 3))
- array([[1, 2, 3], [1, 2, 3], [1, 2, 3]])

```
style_fn = PATH/'style'/'starry_night.jpg'
style_img = open_image(style_fn)
style_img.shape
```

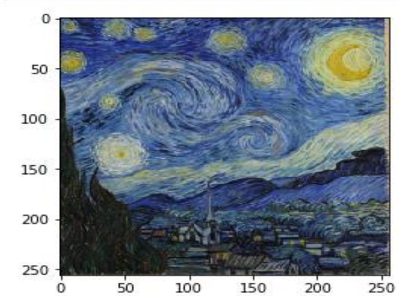
```
(1198, 1513, 3)
```

```
plt.imshow(style_img);
```




```
h,w,_ = style_img.shape
rat = max(sz/h,sz/h)
res = cv2.resize(style_img, (int(w*rat), int(h*rat)), interpolation=cv2.INTER_AREA)
resz_style = res[:sz,-sz:]
```

```
plt.imshow(resz_style);
```




```
style_tfm,_ = tfms[1](resz_style,resz_style)
```

```
style_tfm = np.broadcast_to(style_tfm[None], (bs,)+style_tfm.shape)
```

2.4 总损失

- 合并MSE损失与gram损失
- 注意：之前gram有误，应使用torch.bmm而非torch.mm，因为bmm是每个batch矩阵相乘

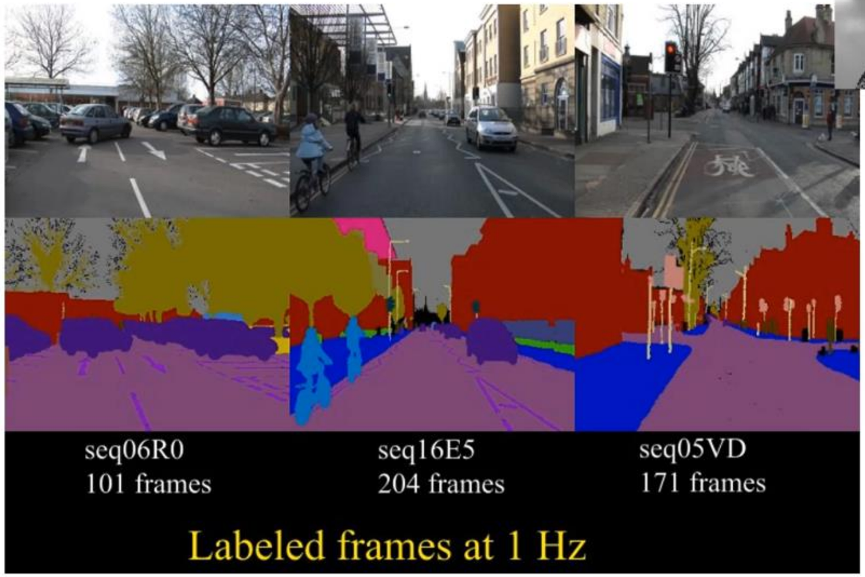
```
class CombinedLoss(nn.Module):
    def __init__(self, m, layer_ids, style_im, ct_wgt, style_wgts):
        super().__init__()
        self.m,self.ct_wgt,self.style_wgts = m,ct_wgt,style_wgts
        self.sfs = [SaveFeatures(m[i]) for i in layer_ids]
        m(VV(style_im))
        self.style_feat = [V(o.features.data.clone()) for o in self.sfs]

    def forward(self, input, target, sum_layers=True):
        self.m(VV(target.data))
        targ_feat = self.sfs[2].features.data.clone()
        self.m(input)
        inp_feat = [o.features for o in self.sfs]

        res = [ct_loss(inp_feat[2],V(targ_feat)) * self.ct_wgt]
        res += [gram_loss(inp,targ)*wgt for inp,targ,wgt
                in zip(inp_feat, self.style_feat, self.style_wgts)]

        if sum_layers: res = sum(res)
        return res

    def close(self):
        for o in self.sfs: o.remove()
```

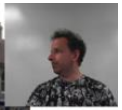
2.5 结果

目录 Contents

3.1 简述

- CamVid数据集
- 标签对应ID映射到颜色
- 每个像素对应一个类型而非每个东西周围找到一个方框





3.2 数据集

- 使用Kaggle竞赛中的Carvana数据集
- 1.统一图像大小并重命名
- 2.分割验证集与训练集（相同的车只能同时出现在一个集中）

3.3 部分代码展示（使用ResNet34）

```
class Empty(nn.Module):
    def forward(self,x): return x

models = ConvnetBuilder(resnet34, 0, 0, 0, custom_head=Empty())
learn = ConvLearner(md, models)
learn.summary()
```

```
class StdUpsample(nn.Module):
    def __init__(self, nin, nout):
        super().__init__()
        self.conv = nn.ConvTranspose2d(nin, nout, 2, stride=2)
        self.bn = nn.BatchNorm2d(nout)

    def forward(self, x): return self.bn(F.relu(self.conv(x)))
```

```
flatten_channel = Lambda(lambda x: x[:,0])
```

```
simple_up = nn.Sequential(
    nn.ReLU(),
    StdUpsample(512,256),
    StdUpsample(256,256),
    StdUpsample(256,256),
    StdUpsample(256,256),
    nn.ConvTranspose2d(256, 1, 2, stride=2),
    flatten_channel
)
```

```
models = ConvnetBuilder(resnet34, 0, 0, 0, custom_head=simple_up)
learn = ConvLearner(md, models)
learn.opt_fn=optim.Adam
learn.crit=nn.BCEWithLogitsLoss()
learn.metrics=[accuracy_thresh(0.5)]
```

```
learn.lr_find()
learn.sched.plot()
```

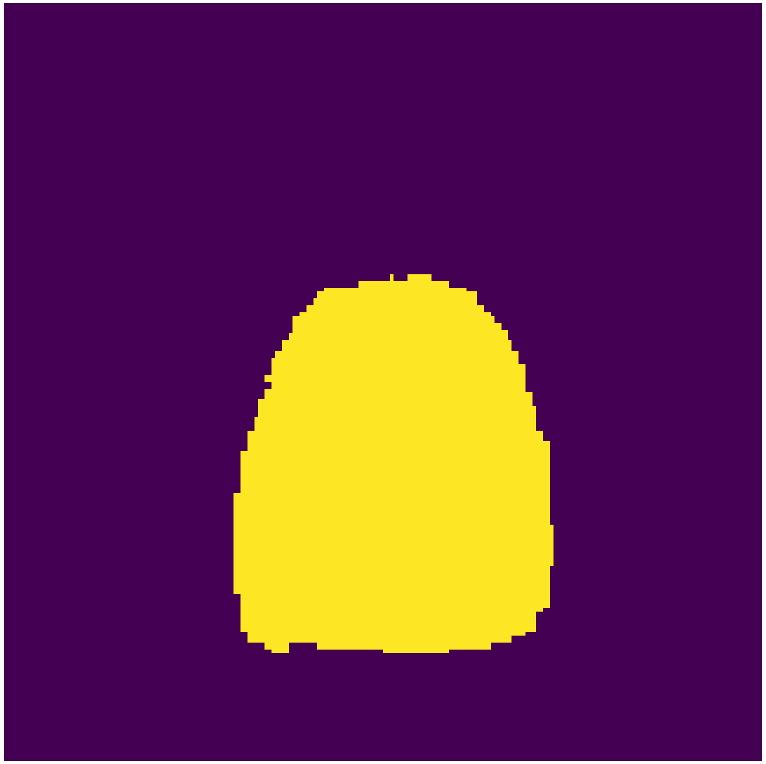
3.4 逐步改进的结果

Only train head		0.96324310824275017
unfreeze		0.99172959849238396
Upscale 512*512		0.99639255659920833
Upscale 1024*1024		0.99817223208291195

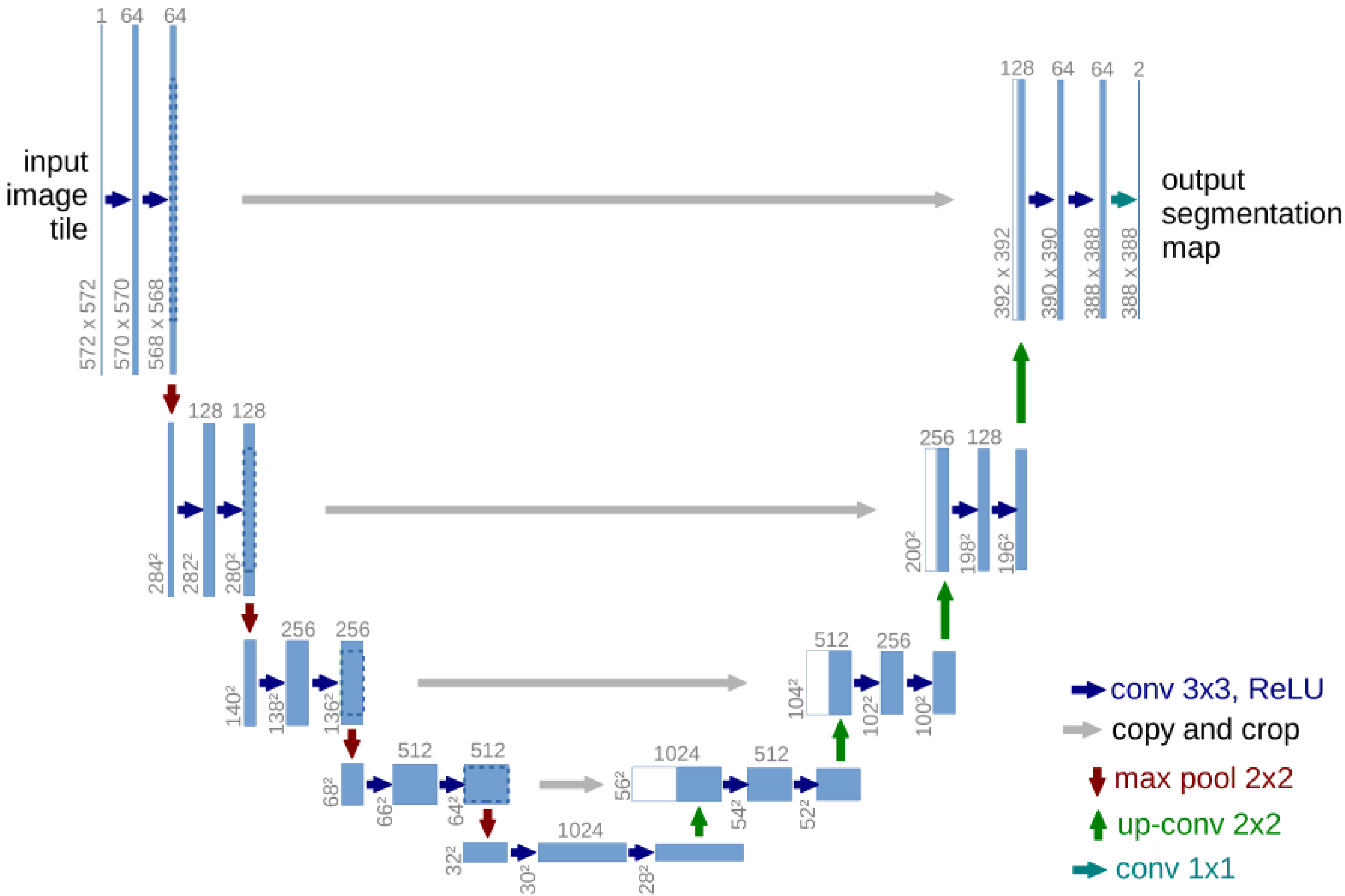
3.5 判断标准

- Dice（Kaggle比赛用的计算方式） 96.8 at 128x128

```
def dice(pred, targs):
    pred = (pred>0).float()
    return 2. * (pred*targs).sum() / (pred+targs).sum()
```



3.6 U-Net



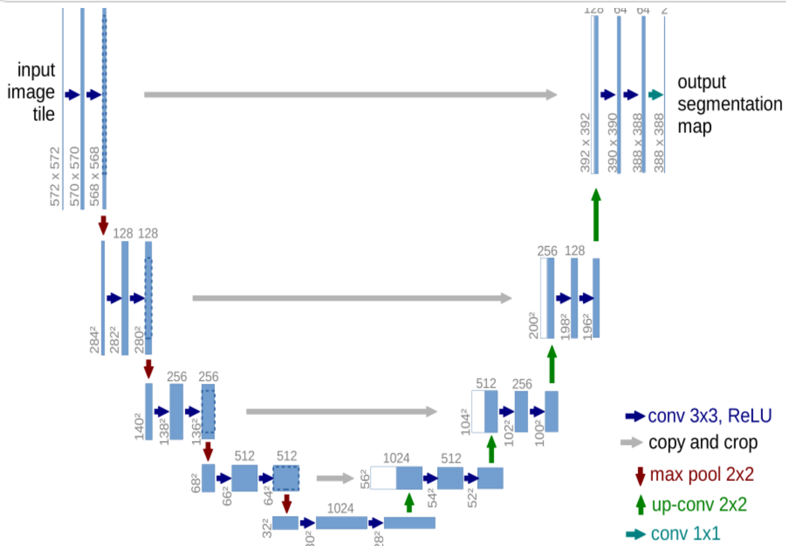
input image tile
output segmentation map
1
64
64
572 x 572
570 x 570
568 x 568
128
128
284²
282²
280²
256
256
140²
138²
136²
512
512
68²
66²
64²
1024
32²
30²
28²
1024
512
56²
54²
52²
512
256
104²
102²
100²
256
128
200²
198²
196²
128
64
64
2
392 x 392
390 x 390
388 x 388
388 x 388
conv 3x3, ReLU
copy and crop
max pool 2x2
up-conv 2x2
conv 1x1

3.6 U-Net

- 左边是不断利用最大池化将图像大小减半，右边则是用反卷积使图像的大小与原图接近，在每一次反卷积后将之前对应大小的特征矩阵复制后与反卷积的结果进行拼接。
- 将所有经过卷积层后池化前的特征信息都保留了，而非仅仅使用被最大池化压缩后的信息

3.6 U-Net

```python
class UnetBlock(nn.Module):
    def __init__(self, up_in, x_in, n_out):
        super().__init__()
        up_out = x_out = n_out//2
        self.x_conv  = nn.Conv2d(x_in,  x_out,  1)
        self.tr_conv = nn.ConvTranspose2d(up_in, up_out, 2, stride=2)
        self.bn = nn.BatchNorm2d(n_out)

    def forward(self, up_p, x_p):
        up_p = self.tr_conv(up_p)
        x_p = self.x_conv(x_p)
        cat_p = torch.cat([up_p,x_p], dim=1)
        return self.bn(F.relu(cat_p))
```



- Up_in之前层所输出的结果
- X_in左边复制过来的层
- N_out输出的数目

3.7 结果

Only train head	0.982704
unfreeze	0.9852004420189631
Upscale 512*512	0.993358936574724
Upscale 1024*1024	0.9959913929776539

谢 谢！