所谓图像风格迁移，是指利用算法学习著名画作的风格，然后再把这种风格应用到另外一张图片上的技术。著名的国像处理应用Prisma是利用风格迁移技术，将普通用户的照片自动变换为具有艺术家的风格的图片。这篇文章会介绍这项技术背后的原理，此外，还会使用TensorFlow 实现一个快速风格迁移的应用。

1 图像风格迁移的原理

1.1 原始图像风格迁移的原理

在学习原始的图像风格迁移之前，可以先回忆一下第二章学习过的ImageNet图像识别模型VGGNet。

事实上，可以这样理解VGGNet的结构：前面的卷积层是从图像中提取“特征”，而后面的全连接层把图片的“特征”转换为类别概率。其中，VGGNet中的浅层（如conv1\_1，conv1\_2），提取的特征往往是比较简单的（如检测点、线、亮度），VGGNet中的深层（如conv5\_1、conv5\_2），提取的特征往往是比较复杂（如有无人脸或某种特定物体）。

VGGNet的本意是输入图像，提取特征，并输出图像类别。图像风格迁移正好与其相反，输入特征，输出对应这种特征的图片，如图7-1所示。

具体来说，风格迁移使用卷积层的中间特征还原出对应这种特征的原始图像。如图7-2a所示，先选取衣服原始图像，经过VGGNet计算后得到各种卷积层特征。接下来，根据这些卷积层的特征，还原出对应这种特征的原始图像。图像b、c 、d 、e 、f 分别为使用卷积层conv1\_2、conv2\_2 、conv3 \_2 、conv4\_2 、conv5\_2 的还原图像。可以发现：浅层的还原效果往往比较好，卷积特征基本保留了所高原始图像中形状、位置、颜色、纹理等信息； 深层对应的还原图像丢失了部分颜色和纹理信息，但大体保留原始图像中物体的形状和位置。

还原图像的方法是梯度下降法。设原始图像为p⃗ \vec{p}*p*​，期望还原的图像为x⃗ \vec{x}*x*（即自动生成的图像）。使用的卷积是第ll*l*层，原始图像p⃗ \vec{p}*p*​在第ll*l*层的特征为PlijP\_{ij}^{l}*Pijl*​。ii*i*表示卷积的第i个通道，j表示卷积的第j个位置。通常卷积的特征是三维的，三维坐标分别对应（高、宽、通道）。此处不考虑具体的高和宽，只考虑位置j，相当于把卷积“压扁”了。比如一个10x10x32的卷积特征，对应1⩽i⩽32,1⩽j⩽1001\leqslant i\leqslant 32,1\leqslant j\leqslant 1001⩽*i*⩽32,1⩽*j*⩽100。对于生成图像x⃗ \vec{x}*x*，同样定义它在ll*l*层的卷积特征为FlijF\_{ij}^{l}*Fijl*​。

有了上面这些符号后，可以写出“内容损失”（Content Loss）。内容损失Lcontent(p⃗ ,x⃗ ,l)L\_{content}(\vec{p},\vec{x},l)*Lcontent*​(*p*​,*x*,*l*)的定义是：  
Lcontent(p⃗ ,x⃗ ,l)=12∑i,j(Flij−Plij)2L\_{content}(\vec{p},\vec{x},l)=\frac{1}{2}\sum \_{i,j}(F\_{ij}^{l}-P\_{ij}^{l})^{2}*Lcontent*​(*p*​,*x*,*l*)=21​∑*i*,*j*​(*Fijl*​−*Pijl*​)2  
Lcontent(p⃗ ,x⃗ ,l)L\_{content}(\vec{p},\vec{x},l)*Lcontent*​(*p*​,*x*,*l*)描述了原始图像p⃗ \vec{p}*p*​和生成图像x⃗ \vec{x}*x*在内容上的“差异”。内容损失越小，说明他们的内容越接近；内容损失越大，说明他们的内容差距也越大。先使用原始图像p⃗ \vec{p}*p*​计算出它的卷积特征PlijP\_{ij}^{l}*Pijl*​，同时随机初始化x⃗ \vec{x}*x*。接着以内容损失Lcontent(p⃗ ,x⃗ ,l)L\_{content}(\vec{p},\vec{x},l)*Lcontent*​(*p*​,*x*,*l*)为优化目标，通过梯度下降法逐步改变x⃗ \vec{x}*x*，进过一定的步数后，得到的x⃗ \vec{x}*x*是希望的还原图像了。在这个过程中，内容损失Lcontent(p⃗ ,x⃗ ,l)L\_{content}(\vec{p},\vec{x},l)*Lcontent*​(*p*​,*x*,*l*)应该是越来越小了。

除了还原图像原本的“内容”之外，另一方面，还希望还原图像的“风格”。那么图像的“风格”应该用什么来表示呢？一种方法是使用图像的卷积层特征的Gram矩阵。

Gram矩阵是关于一组向量的内积的对称矩阵，例如，向量组x1⃗ ,x2⃗ ,...,xn⃗ \vec{x\_1},\vec{x\_2},...,\vec{x\_n}*x*1​​,*x*2​​,...,*xn*​​的Gram矩阵是：

通常取内积为欧几里得空间上的标准内积， 即(xi⃗ ,xj⃗ )=xi⃗ Txj⃗ (\vec{x\_i}, \vec{x\_j})=\vec{x\_i}^{T}\vec{x\_j}(*xi*​​,*xj*​​)=*xi*​​*Txj*​​。

设卷积层的输出为FlijF\_{ij}^{l}*Fijl*​，那么这个卷积特征对应的Gram矩阵的第i行第j个元素定义为  
Glij=∑kFlikFljkG\_{ij}^{l}=\sum \_{k}F\_{ik}^{l}F\_{jk}^{l}*Gijl*​=∑*k*​*Fikl*​*Fjkl*​  
设在第l层中，卷积特征的通道数为NlN\_{l}*Nl*​，卷积的高、宽成绩数为MlM\_l*Ml*​，那么FlijF\_{ij}^{l}*Fijl*​满足1⩽i⩽Nl,1⩽i⩽Ml1\leqslant i\leqslant N\_l,1\leqslant i\leqslant M\_l1⩽*i*⩽*Nl*​,1⩽*i*⩽*Ml*​。G实际是向量组Fl1,Fl2,⋅⋅⋅Fli,⋅⋅⋅FlNlF\_1^{l},F\_2^{l},\cdot \cdot \cdot F\_i^{l},\cdot \cdot \cdot F\_{N\_l}^{l}*F*1*l*​,*F*2*l*​,⋅⋅⋅*Fil*​,⋅⋅⋅*FNl*​*l*​的Gram矩阵，其中Fli=(Fli1,Fli2,⋅⋅⋅Flij,⋅⋅⋅FliMl)F\_i^{l} = (F\_{i1}^{l},F\_{i2}^{l},\cdot \cdot \cdot F\_{ij}^{l},\cdot \cdot \cdot F\_{iM\_l}^{l})*Fil*​=(*Fi*1*l*​,*Fi*2*l*​,⋅⋅⋅*Fijl*​,⋅⋅⋅*FiMl*​*l*​)

此处数学符号特较多，因此再举一个例子来加深读者对此Gram矩阵的理解。假设某一层输出的卷积特征为10x10x32，即它是一个宽、高均为10，通道数为32的张量。Fl1F\_1^{l}*F*1*l*​表示第一个通道的特征，它是一个100维的向量。Fl2F\_2^{l}*F*2*l*​表示第二个通道的特征，他同样是一个100维的向量，它对应的Gram矩阵G。

Gram矩阵可以在一定程度上反映原始图像的“风格”。仿照“内容损失”，还可以定义一个“风格损失”（Style Loss）。设原始图像为a⃗ \vec {a}*a*，要还原的风格图像为x⃗ \vec {x}*x*，先计算出原始图像某一层卷积的Gram矩阵为AlA^{l}*Al*，要还原的图像x⃗ \vec {x}*x*经过同样的计算得到对应卷积层的Gram矩阵是GlG^{l}*Gl*，风格损失定义为  
Lstyle(p⃗ ,x⃗ ,l)=14N2lM2l∑i,j(Alij−Glij)2L\_{style}(\vec{p},\vec{x},l)=\dfrac{1}{4N\_{l}^{2}M\_{l}^{2}}\sum \_{i,j}(A\_{ij}^{l}-G\_{ij}^{l})^{2}*Lstyle*​(*p*​,*x*,*l*)=4*Nl*2​*Ml*2​1​∑*i*,*j*​(*Aijl*​−*Gijl*​)2  
分母上的4N2lM2l4N\_{l}^{2}M\_{l}^{2}4*Nl*2​*Ml*2​是一个归一化项，目的是防止风格损失的数量级相比内容损失过大。在实际应用中，常利用多层而非一层的风格损失，多层的风格损失是单层风格损失的加权累加，即Lstyle(p⃗ ,x⃗ )=∑lwlLstyle(p⃗ ,x⃗ ,l)L\_{style}(\vec{p},\vec{x})=\sum \_{l}w\_lL{style}(\vec{p},\vec{x},l)*Lstyle*​(*p*​,*x*)=∑*l*​*wl*​*Lstyle*(*p*​,*x*,*l*)，其中wlw\_l*wl*​表示第ll*l*层的权重。

利用风格损失，可以还原出图像的风格了，如图7-3所示，尝试还原梵高的著名画作《星空》的风格。

其中，图7-3b是由conv1\_1的风格损失还原的，图7-3c是由conv1\_1，conv2\_1两层的风格损失还原的，图7-3d是由conv1\_1，conv2\_1，conv3\_1，图7-3e为conv1\_1，conv2\_1，conv3\_1，conv4\_1风格损失还原的。使用浅层还原的“风格图像”的纹理尺度往往比较小，只保留了颜色和局部的纹理（如图7-3b）；组合深层、浅层还原出的“风格图像”更加真实且接近原图片（如图7-3f）。  
总结一下，到目前为止介绍了两个内容：

* ****利用内容损失还原图像内容****
* ****利用风格损失还原图像风格****

****那么，可不可以将内容损失和风格损失组合起来，在还原一张图像的同事还原里一张图像的风格呢？答案是肯定的，这是图像风格迁移的基本算法。****

设原始的内容图像为p⃗ \vec {p}*p*​，原始的风格图像为a⃗ \vec {a}*a*，待生成的图像为x⃗ \vec {x}*x*。希望x⃗ \vec {x}*x*可以保持内容图像p⃗ \vec {p}*p*​的内容，同事具备风格图像a⃗ \vec {a}*a*的风格。因此组合p⃗ \vec {p}*p*​的内容损失和a⃗ \vec {a}*a*的风格损失，定义总的损失函数为：  
Ltotal(p⃗ ,a⃗ ,x⃗ )=αLcontent(p⃗ ,x⃗ )+βLstyle(a⃗ ,x⃗ )L\_{total}(\vec{p},\vec{a},\vec{x})=\alpha L\_{content}(\vec{p},\vec{x})+\beta L\_{style}(\vec{a},\vec{x})*Ltotal*​(*p*​,*a*,*x*)=*αLcontent*​(*p*​,*x*)+*βLstyle*​(*a*,*x*)  
α,β\alpha ,\beta*α*,*β*是平衡两个损失的超参数。如果α\alpha*α*偏大，还原的图像会更接近a⃗ \vec {a}*a*的风格。使用总的损失函数可以组合p⃗ \vec {p}*p*​的内容和a⃗ \vec {a}*a*的风格，这实现了图像风格的迁移。部分还原的图像如图7-4所示。

以上是原始的图像风格迁移的基本原理。事实上，原始图像迁移的速度非常慢，在CPU上生成一张图片需要数十分钟甚至几个小时，即使在GPU上也需要数分钟才能生成一张较大的图片，这大大限制了这项技术的使用场景。速度慢的原因在于，要用总损失Ltotal(p⃗ ,a⃗ ,x⃗ )L\_{total}(\vec{p},\vec{a},\vec{x})*Ltotal*​(*p*​,*a*,*x*)优化图片x⃗ \vec {x}*x*，这意味着生成一张图片需要几百步梯度下降的迭代，而每一步迭代都需要耗费大量的时间。从另一个角度看，优化x⃗ \vec {x}*x*可以看作是一个“训练模型”的过程，以往都是针对模型参数训练，而这里训练的目标是图片x⃗ \vec {x}*x*，而训练模型一般都比执行训练好的模型要慢得多。下一节将会讲到更实用的快速图像风格迁移，它把原来的“训练”的过程变成了一个“执行”的过程，因此大大加快了生成风格化图片的速度。

## **1.2 快速图像风格迁移的原理**

原始的图像风格迁移用一个损失Ltotal(p⃗ ,a⃗ ,x⃗ )L\_{total}(\vec{p},\vec{a},\vec{x})*Ltotal*​(*p*​,*a*,*x*)来衡量x⃗ \vec {x}*x*是否成功组合了p⃗ \vec {p}*p*​的内容和a⃗ \vec {a}*a*的风格。然后以Ltotal(p⃗ ,a⃗ ,x⃗ )L\_{total}(\vec{p},\vec{a},\vec{x})*Ltotal*​(*p*​,*a*,*x*)为目标，用梯度下降法来逐步迭代x⃗ \vec {x}*x*。因为在生成图像过程中需要逐步对x⃗ \vec {x}*x*优化，所以速度很慢。

快速图像风格迁移的方法是：不使用优化的方法来逐步迭代生成x⃗ \vec {x}*x*，而是使用一个神经网络直接生成x⃗ \vec {x}*x*。对应的网络结构如图7-5所示。

整个系统由两个神经网络组成，它们在图中由连个虚线框分别标出。左边的是图像生成网络，右边是损失网络。损失网络实际上是VGGNet，这与原始的风格迁移是一致的。同1.1节一样，利用损失网络来定义内容损失、风格损失。这个损失用来训练图像生成网络。图像生成网络的职责是生成某一种风格的图像，它的输入是一个图像，输出同样是一个图像。由于生成图像只需要在生成网络中计算一遍，所以速度比原始图像风格迁移快很多。

同样使用数学符号严格地阐述上面的过程：设输入的图像为x⃗ \vec {x}*x*，经过图像生成网络生成的图像为y⃗ \vec {y}*y*​。y⃗ \vec {y}*y*​在内容上应该与原始的内容图像yc⃗ \vec {y\_c}*yc*​​接近，因此可以利用损失网络定义内容损失Lcontent(y⃗ ,yc⃗ )L\_{content}(\vec {y}, \vec {y\_c})*Lcontent*​(*y*​,*yc*​​)，内容损失使用的是VGG-16的relu3\_3层输出的特征，对应图7-5中的lϕ,relu3\_3featl\_{feat}^{\phi,relu3\\_3}*lfeatϕ*,*relu*3\_3​。另一方面，我们还希望y⃗ \vec {y}*y*​具有目标风格图像ys⃗ \vec {y\_s}*ys*​​的风格，因此又可以定义一个风格损失Ltotal(y⃗ ,yc⃗ ,ys⃗ )L\_{total}(\vec {y},\vec {y\_c}, \vec {y\_s})*Ltotal*​(*y*​,*yc*​​,*ys*​​)。定义风格损失时使用了VGG-16的四个中检出呢个relu1\_2，relu2\_2，relu3\_3，relu4\_3，对应图中的lϕ,relu1\_2feat,lϕ,relu2\_2feat,lϕ,relu3\_3feat,lϕ,relu4\_3featl\_{feat}^{\phi,relu1\\_2},l\_{feat}^{\phi,relu2\\_2},l\_{feat}^{\phi,relu3\\_3},l\_{feat}^{\phi,relu4\\_3}*lfeatϕ*,*relu*1\_2​,*lfeatϕ*,*relu*2\_2​,*lfeatϕ*,*relu*3\_3​,*lfeatϕ*,*relu*4\_3​。同样组合这两个损失得到一个总损失Ltotal(y⃗ ,yc⃗ ,ys⃗ )L\_{total}(\vec {y},\vec {y\_c},\vec {y\_s})*Ltotal*​(*y*​,*yc*​​,*ys*​​)。利用总损失可以训练图像生成网络。训练完成后直接使用图像生成网络生成图像。值得一提的是，在整个训练过程中，一般只是固定使用一种风格ys⃗ \vec {y\_s}*ys*​​，而图像内容yc⃗ \vec {y\_c}*yc*​​取和输入x⃗ \vec {x}*x*一样，即ys⃗ =x⃗ \vec {y\_s}=\vec {x}*ys*​​=*x*。

# **2 在TensorFlow 中实现快速风格迁移**

在本节中，首先会介绍代码结构，然后讲解如何使用预训练的模型，以及如何自己训练模型，最后说明该项目的一些实现细节。

该项目建立在另一个开源项目TensorFlow Slim的基础上，TensorFlow Slim是基于TensorFlow的一个开源图像分类库，它定义了常用的ImageNet模型。而其中的VGG16模型正式在定义损失网络时要用到的。

在运行项目前，请确保使用python 2.7版本和TensorFlow大于1.0的版本。并需要安装pyyaml库，安装方法是：

pip install pyyaml

## **2.1 使用预训练模型**

在chapter\_7\_data/中提供了7个预训练模型：wave.ckpt-done、cubist.ckpt-done、denoised\_starry.skpt-done、mosaic.ckpt-done、scream.ckpt-done、feathers. ckpt-done。回到源码目录chapter\_7/，在其中新建一个model文件夹，然后把需要使用的模型文件复制到这个文件夹models/wave.ckpt-done。接下来运行下面的命令可以生成一张风格化图像了：

python eval.py --model\_file models/wave.ckpt-done --image\_file img/test.jpg

–model\_file后面指定了与训练的模型的文件位置。如果没有把预训练模型保存为models/wave.ckpt-done，也可以自行替换为相应的文件位置。–image\_file表示需要进行风格化的图像，在这里指定的是img目录下名为test.jpg 的示例图像），也可以使用自己的图像进行尝试，同样只需要指定合适的文件位置即可。

运行上述命令后，成功风格化的图像会被写到generated/res.jpg 。我们可以打开该文件进行查看。除了模型wave.ckpt-done ，还可以运行其他的预训练、模型。七个预训练模型及其风格化图片效果见表7-3 。

## **2.2 训练自己的模型**

如何训练自己的图像生成模型呢？这里以wave为例，介绍训练模型的全过程。

在训练之前，需要完成两项前期准备工作。首先下载VGG16模型，将下载到的压缩包解压后会得到一个[vgg16.ckpt文件](http://download.tensorflow.org/models/vgg_16_2016_08_ 28.tar.gz)。在chapter\_7/中新建一个文件夹pretrained，并将vgg16.ckpt复制到pretrained文件夹中。最后文件的路径是pretrained/vgg16.ckpt。另外需要[下载COCO数据集](http://msvocds.blob.core.windows.net/coco2014/train2014.zip)。将该数据集解压后会得到一个train2014文件夹，其中应该含有大量jpg格式的图片。Windows用户请将该文件夹移动到chapter\_7/中。Linux用户可以不用移动，只要在chapter\_7/中使用下面的命令，简历到train2014文件夹的符号连接可以了：

ln -s <到train2014文件夹的路径> train2014

接下来可以训练模型了。以模型wave为例，对应的训练命令是：

python train.py -c conf/wave.yml

该命令放入含义是利用已经写好的conf/wave.yml文件来训练模型。wave.yml为配置文件，内容为：

## Basic configuration 基础配置 style\_image: img/wave.jpg # targeted style image 指定原始的风格图像 # 这个模型的名字，一般和图像名称保持一致。这个名字决定了checkpoint和events文件的保存文件夹 naming: "wave" # the name of this model. Determine the path to save checkpoint and events file. # checkpoint和events文件保存的根目录。最后所有的checkpoint和events文件会被保存在<model\_path>/<naming>下 model\_path: models # root path to save checkpoint and events file. The final path would be <model\_path>/<naming> ## Weight of the loss 各损失权重 content\_weight: 1.0 # weight for content features loss内容损失的权重 style\_weight: 220.0 # weight for style features loss风格损失权重 # 损失的权重。这是原论文中提到的一个损失。在这个项目中发现设定它的权重为0也不会影响收敛，所以没有提及 tv\_weight: 0.0 # weight for total variation loss ## The size, the iter number to run image\_size: 256 #训练原始图像大小 batch\_size: 4 #一次batch的样本数 epoch: 2 # 跑的epoch的运行次数 ## Loss Network 损失网络 loss\_model: "vgg\_16" #使用vgg\_16模型 content\_layers: # use these layers for content loss使用conv3\_3定义内容损失 - "vgg\_16/conv3/conv3\_3" style\_layers: # use these layers for style loss使用这些卷积层定义风格损失 - "vgg\_16/conv1/conv1\_2" - "vgg\_16/conv2/conv2\_2" - "vgg\_16/conv3/conv3\_3" - "vgg\_16/conv4/conv4\_3" checkpoint\_exclude\_scopes: "vgg\_16/fc" # we only use the convolution layers, so ignore fc layers.我们只用到卷积层，所以不需要fc层 loss\_model\_file: "pretrained/vgg\_16.ckpt" # the path to the checkpoint # 预训练模型vgg\_16.ckpt对应的位置

如果我们希望训练新的“风格”，可以选取一张风格图片，并编写新的yml配置文件。其中，需要把style\_image修改为新图片所在位置，并修改对应的naming。这样就可以进行训练了。最后，可以使用训练完成的check【oint生成图片。杂训练新的“风格”时，有可能会需要调整个个损失之间的权重。调整的方法在下一节中进行叙述。

## **2.3 在TensorBoard中监控训练情况**

在训练过程中，可以打开TensorBoard监控训练情况。仍以wave模型为例：

tensorboard --logdir models/wave/

访问http://localhost:6006即可打开Tensorboard的主页面。训练时最先关心的应该是损失损失下降的情况。损失主要由风格损失、内容损失两项构成。展开loss选项卡可以看到损失的变化情况，如图7-7所示：

center\_loss和style\_loss分别对应了内容损失和风格损失，中间的regularizer\_loss可以暂时不用理会。最理想的情况是content\_loss和style\_loss随着训练地不断下降。在训练的初期可能会出现只有style\_loss下降而content\_loss上升的情况，不过这是暂时的，最后两个损失都会出现较为稳定的下降。

当训练新的“风格”时，再时可能还会需要调整配置文件中的content\_ weight 和style\_ weight 。当content\_weight 过大时，观察到的generated图像会非常接近原始的origin图像。而style\_ weight 过大时，会导致图像过于接近原始的风格图像，此时的generated 图像如图7-9 所示，几乎看不到origin图像的内容。在训练时，需要合理调整style\_weight和content\_weight的比重。

## **2.4 项目实现细节**

最后讨论项目的细节。该项目使用了两个网络，即损失网络与生成网络。损失网络为VGG16模型，用的是TensorFlow Slim中已经写好的代码，图像生成网络可以自己进行定义。

****1. 损失网络、图像生成网络的定义与引用****  
损失网络使用TenorFlow Slim 的VGG16模型， 它的实际定义位置是在nets/vgg.py文件中，不过没再必要知道它的详细源码，只需要了解是如何在  
训练过程中引用它的。

[而图像生成网络在models.py](http://xn--models-2q1j753b3kat44gnp0c1oyapibx9d.py/) 中定义，它的关键代码如下：

# 定义图像生成网络 def net(image, training): # 一开始在图片的上下左右加上一些额外的“边框”，目的是消除边缘效应 # Less border effects when padding a little before passing through .. image = tf.pad(image, [[0, 0], [10, 10], [10, 10], [0, 0]], mode='REFLECT') # 三层卷积层 with tf.variable\_scope('conv1'): conv1 = relu(instance\_norm(conv2d(image, 3, 32, 9, 1))) with tf.variable\_scope('conv2'): conv2 = relu(instance\_norm(conv2d(conv1, 32, 64, 3, 2))) with tf.variable\_scope('conv3'): conv3 = relu(instance\_norm(conv2d(conv2, 64, 128, 3, 2))) # 仿照ResNet定义一些跳过连接 with tf.variable\_scope('res1'): res1 = residual(conv3, 128, 3, 1) with tf.variable\_scope('res2'): res2 = residual(res1, 128, 3, 1) with tf.variable\_scope('res3'): res3 = residual(res2, 128, 3, 1) with tf.variable\_scope('res4'): res4 = residual(res3, 128, 3, 1) with tf.variable\_scope('res5'): res5 = residual(res4, 128, 3, 1) # 定义卷积之后定义反卷积 # 反卷积不采用通常的转置卷积的方式，而是采用先放大，在做卷积的方式 # print(res5.get\_shape()) with tf.variable\_scope('deconv1'): # deconv1 = relu(instance\_norm(conv2d\_transpose(res5, 128, 64, 3, 2))) deconv1 = relu(instance\_norm(resize\_conv2d(res5, 128, 64, 3, 2, training))) with tf.variable\_scope('deconv2'): # deconv2 = relu(instance\_norm(conv2d\_transpose(deconv1, 64, 32, 3, 2))) deconv2 = relu(instance\_norm(resize\_conv2d(deconv1, 64, 32, 3, 2, training))) with tf.variable\_scope('deconv3'): # deconv\_test = relu(instance\_norm(conv2d(deconv2, 32, 32, 2, 1))) deconv3 = tf.nn.tanh(instance\_norm(conv2d(deconv2, 32, 3, 9, 1))) # decanv3是经过tanh函数得到的输出值，所以它的值域范围是-1~1 # 知道RGB图像的像素范围是0~255，所以这里对deconv3进行这样的缩放 y = (deconv3 + 1) \* 127.5 # Remove border effect reducing padding. # 最后取出一开始为了防止边缘效应而加入的“边框” height = tf.shape(y)[1] width = tf.shape(y)[2] y = tf.slice(y, [0, 10, 10, 0], tf.stack([-1, height - 20, width - 20, -1])) return y

图像生成网络的原理主要是先对图像卷积计算，然后再进行“反卷积”计算。相当于对图像进行编码，然后再还原为图像。在“反卷积”的过程中，一般使用转置卷积，但在这里可能会导致一些堆叠噪声。此处，使用resize\_conv2d来代替转置卷积，它的原理是先对图片放大，然后再进行卷积计算。此外，还有一些提高图像质量的小技巧。比如使用所谓的instance  
normalization 代替常用的batch normalization 。关于instance normalization 、转置卷积的详细原理，可以参阅相关资料进行了解，这里不再详细展  
开了。

定义好图像生成网络和损失网络后，可以在训练时引用。[相应的代码在train.py](http://xn--train-ck2h370ci5n1z4cd7an2f.py/) 文件中：

# network\_fn是损失网络的函数。因为不需要对损失函数训练，所以is\_training = False network\_fn = nets\_factory.get\_network\_fn( FLAGS.loss\_model, num\_classes=1, is\_training=False) # 损失网络中要用的图像的预处理函数 image\_preprocessing\_fn, image\_unprocessing\_fn = preprocessing\_factory.get\_preprocessing( FLAGS.loss\_model, is\_training=False) # 读入训练图像 processed\_images = reader.image(FLAGS.batch\_size, FLAGS.image\_size, FLAGS.image\_size, 'train2014/', image\_preprocessing\_fn, epochs=FLAGS.epoch) # 此处引用图像生成网络。model.net是图像生成网络,generated是生成的图像 # 设置training = True，因为要训练该网络 generated = model.net(processed\_images, training=True) # 将生成的图像generated同样使用image\_preprocessing\_fn进行处理 # 因为generated同样需要送到损失网络中计算loss processed\_generated = [image\_preprocessing\_fn(image, FLAGS.image\_size, FLAGS.image\_size) for image in tf.unstack(generated, axis=0, num=FLAGS.batch\_size) ] processed\_generated = tf.stack(processed\_generated) # 将原始图像、生成图像送到损失网络中 # 这里将它们合并后再送到网络中计算，因为同一的计算可以加快速度 # 将原始图像、生成图像送到损失网络并计算后，将使用结果endpoints\_dict 计算损失 \_, endpoints\_dict = network\_fn(tf.concat([processed\_generated, processed\_images], 0), spatial\_squeeze=False)

1. ****内容损失和风格损失的定义****  
   损失的定义基本由文件loss .py中的函数完成。先来介绍如何定义内容损失：
2. # endpoints\_dict是上一节提到的损失网络各层的计算结果 # content\_layers是定义使用哪些层的差距计算损失，默认配置是conv3\_3 def content\_loss(endpoints\_dict, content\_layers): content\_loss = 0 for layer in content\_layers: # 上一节中把生成的图像、原始图像同时传入损失网络中计算 # 所以这里先把他们区分开 # 读者可以参照函数tf.concat与tf.split的文档理解此处的内容 generated\_images, content\_images = tf.split(endpoints\_dict[layer], 2, 0) size = tf.size(generated\_images) # 所谓的内容损失，是生成图片generated\_images与原始图片激活content\_images的L\*L距离 content\_loss += tf.nn.l2\_loss(generated\_images - content\_images) \* 2 / tf.to\_float(size) # remain the same as in the paper return content\_loss
3. 在看如何定义风格损失：
4. # 定义风格损失 # style\_layers为定义使用哪些层计算风格损失。默认为conv1\_2、conv2\_2、conv3\_3、conv4\_3 # style\_features\_t是利用原始的风格图片计算的层的激活 # 如在wave模型中是img/wave.jpg计算的激活 def style\_loss(endpoints\_dict, style\_features\_t, style\_layers): style\_loss = 0 # summary是为TensorBoard服务的 style\_loss\_summary = {} for style\_gram, layer in zip(style\_features\_t, style\_layers): # 计算风格损失，只需要计算生成图片generated\_imgs与目标风格 # style\_features\_t的差距。因此不需要取出content\_images generated\_images, \_ = tf.split(endpoints\_dict[layer], 2, 0) size = tf.size(generated\_images) # 调用gram函数计算Gram矩阵。风格损失定义为生成图片与目标风格Gram矩阵的L\*L的Loss layer\_style\_loss = tf.nn.l2\_loss(gram(generated\_images) - style\_gram) \* 2 / tf.to\_float(size) style\_loss\_summary[layer] = layer\_style\_loss style\_loss += layer\_style\_loss return style\_loss, style\_loss\_summary
5. 在train.py中，直接利用上面的函数可以得到总的损失：
6. """Build Losses""" # 定义内容损失 content\_loss = losses.content\_loss(endpoints\_dict, FLAGS.content\_layers) # 定义风格损失 style\_loss, style\_loss\_summary = losses.style\_loss(endpoints\_dict, style\_features\_t, FLAGS.style\_layers) # 定义tv损失，该损失在实际训练中并没有被应道，因为在训练时都采用tv\_weight=0 tv\_loss = losses.total\_variation\_loss(generated) # use the unprocessed image # 总损失是这些损失的加权和，最后利用总损失优化图像生成网络即可 loss = FLAGS.style\_weight \* style\_loss + FLAGS.content\_weight \* content\_loss + FLAGS.tv\_weight \* tv\_loss
7. ****3. 确定训练、保存的变量****  
   在本项目中，****只需要训练圄像生成网络中的变量，而不需要训练损失网  
   络中的变量****。在把模型保存成checkpoint时，也只需要保存图像生成网络中的变量。TenorFlow会默认训练、保存所有变量，因此必须把需要训练和需要保存的变量找出来，这也是本项目中的一个注意点。对应的代码同样在  
   train.py中：
8. # 找出需要训练的变量 variable\_to\_train = [] # 使用tf.trainable\_variables()找出所有可训练的变量 for variable in tf.trainable\_variables(): # 如果不在损失网络中，把它们加入列表variable\_to\_train if not(variable.name.startswith(FLAGS.loss\_model)): variable\_to\_train.append(variable) # 定义训练步骤时指定var\_list=variable\_to\_train。这样不会训练损失网络 train\_op = tf.train.AdamOptimizer(1e-3).minimize(loss, global\_step=global\_step, var\_list=variable\_to\_train) # 找出所有需要保存的变量 variables\_to\_restore = [] # 用tf.global\_variable()找出所有变量 for v in tf.global\_variables(): # 不在损失网络中则加入列表variables\_to\_restore if not(v.name.startswith(FLAGS.loss\_model)): variables\_to\_restore.append(v) # 定义saver时指定只会保存variables\_to\_restore saver = tf.train.Saver(variables\_to\_restore, write\_version=tf.train.SaverDef.V1)

# **3 总结**

这篇文章首先详细介绍了原始图像风格迁移的基本原理，其中内容损失、风格损失两种损失函数的定义尤为关键。接着介绍了快速图像风格迁移的原理，并学习了如何使用TenorFlow实现快速图像风格迁移。最后一起研究了项目中的一些实现细节。相信通过这篇文章的介绍，我们可以基本掌握风格迁移这一领域的基本思想与TenorFlow中相应的实现方法。