

## 样式迁移

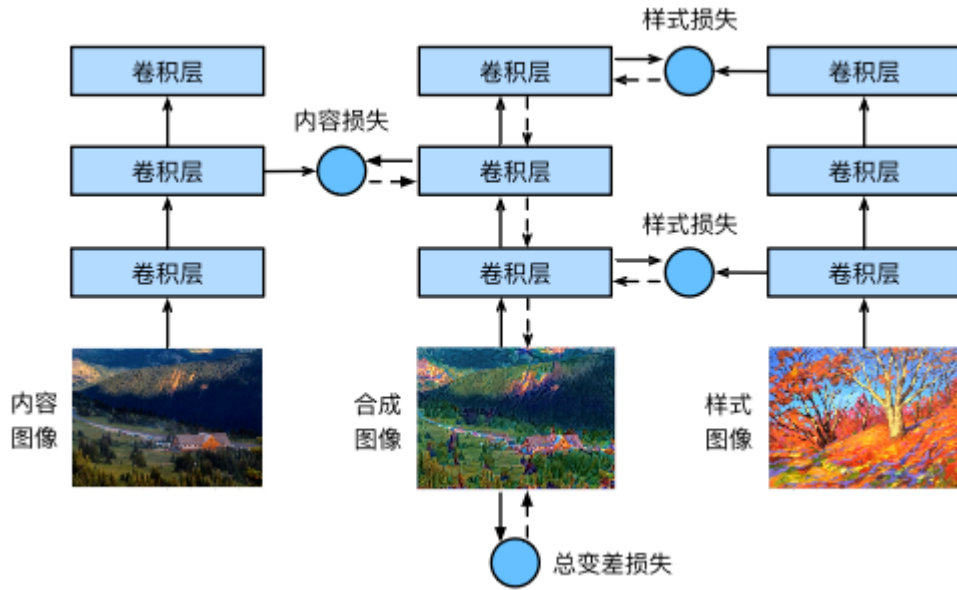
如果你是一位摄影爱好者，也许接触过滤镜。它能改变照片的颜色样式，从而使风景照更加锐利或者令人像更加美白。但一个滤镜通常只能改变照片的某个方面。如果要照片达到理想中的样式，经常需要尝试大量不同的组合，其复杂程度不亚于模型调参。

在本节中，我们将介绍如何使用卷积神经网络自动将某图像中的样式应用在另一图像之上，即样式迁移（style transfer）[1]。这里我们需要两张输入图像，一张是内容图像，另一张是样式图像，我们将使用神经网络修改内容图像使其在样式上接近样式图像。图9.12中的内容图像为本书作者在西雅图郊区的雷尼尔山国家公园（Mount Rainier National Park）拍摄的风景照，而样式图像则是一副主题为秋天橡树的油画。最终输出的合成图像在保留了内容图像中物体主体形状的情况下应用了样式图像的油画笔触，同时也让整体颜色更加鲜艳。



## 方法

图9.13用一个例子来阐述基于卷积神经网络的样式迁移方法。首先，我们初始化合成图像，例如将其初始化成内容图像。该合成图像是样式迁移过程中唯一需要更新的变量，即样式迁移所需迭代的模型参数。然后，我们选择一个预训练的卷积神经网络来抽取图像的特征，其中的模型参数在训练中无须更新。深度卷积神经网络凭借多个层逐级抽取图像的特征。我们可以选择其中某些层的输出作为内容特征或样式特征。以图9.13为例，这里选取的预训练的神经网络含有3个卷积层，其中第二层输出图像的内容特征，而第一层和第三层的输出被作为图像的样式特征。接下来，我们通过正向传播（实线箭头方向）计算样式迁移的损失函数，并通过反向传播（虚线箭头方向）迭代模型参数，即不断更新合成图像。样式迁移常用的损失函数由3部分组成：内容损失（content loss）使合成图像与内容图像在内容特征上接近，样式损失（style loss）令合成图像与样式图像在样式特征上接近，而总变差损失（total variation loss）则有助于减少合成图像中的噪点。最后，当模型训练结束时，我们输出样式迁移的模型参数，即得到最终的合成图像。



下面，我们通过实验来进一步了解样式迁移的技术细节。实验需要用到一些导入的包或模块。

```
In [1]:

%matplotlib inline
import time
import torch
import torch.nn.functional as F
import torchvision
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image

import sys
sys.path.append("/home/kesci/input")
import d2len1909 as d2l
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu') # 均已测试

print(device, torch.__version__)

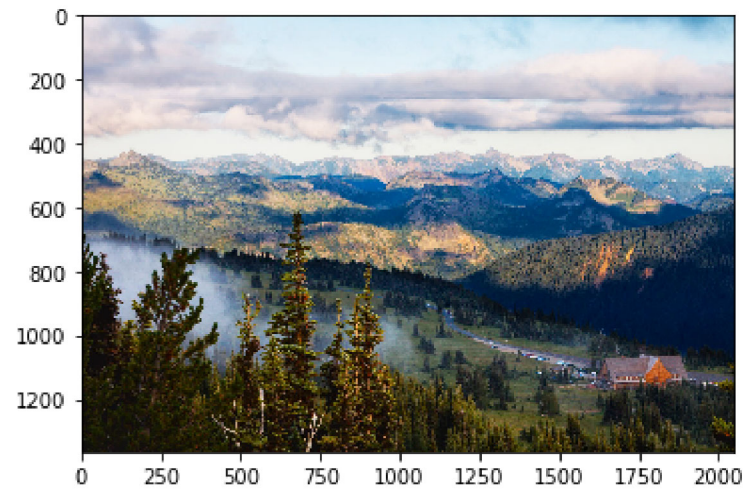
cpu 1.3.0
```

## 读取内容图像和样式图像

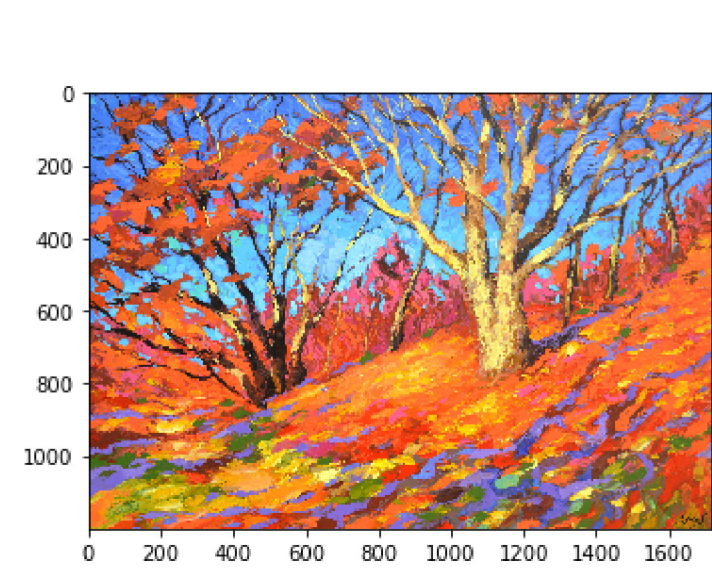
首先，我们分别读取内容图像和样式图像。从打印出的图像坐标轴可以看出，它们的尺寸并不一样。

```
In [2]:

#d2l.set_figsize()
content_img = Image.open('/home/kesci/input/NeuralStyle6163/rainier.jpg')
plt.imshow(content_img);
```



```
In [3]:
style_img = Image.open('/home/kesci/input/NeuralStyle6163/autumn_oak.jpg')
plt.imshow(style_img);
```



## 预处理和后处理图像

下面定义图像的预处理函数和后处理函数。预处理函数`preprocess`对输入图像在RGB三个通道分别做标准化，并将结果变换成卷积神经网络接受的输入格式。后处理函数`postprocess`则将输出图像中的像素值还原回标准化之前的值。由于图像打印函数要求每个像素的浮点数值在0到1之间，我们使用`clamp`函数对小于0和大于1的值分别取0和1。

```
In [4]:

rgb_mean = np.array([0.485, 0.456, 0.406])
rgb_std = np.array([0.229, 0.224, 0.225])

def preprocess(PIL_img, image_shape):
    process = torchvision.transforms.Compose([
        torchvision.transforms.Resize(image_shape),
        torchvision.transforms.ToTensor(),
        torchvision.transforms.Normalize(mean=rgb_mean, std=rgb_std)])

    return process(PIL_img).unsqueeze(dim = 0) # (batch_size, 3, H, W)

def postprocess(img_tensor):
    inv_normalize = torchvision.transforms.Normalize(
        mean= -rgb_mean / rgb_std,
        std= 1/rgb_std)
    to_PIL_image = torchvision.transforms.ToPILImage()
    return to_PIL_image(inv_normalize(img_tensor[0].cpu()).clamp(0, 1))
```

## 抽取特征

我们使用基于ImageNet数据集预训练的VGG-19模型来抽取图像特征 [1]。

```
In [5]:

!echo $TORCH_HOME # 将会把预训练好的模型下载到此处(没有输出的话默认是.cache/torch)
pretrained_net = torchvision.models.vgg19(pretrained=False)
pretrained_net.load_state_dict(torch.load('/home/kesci/input/vgg198261/vgg19-dcbb9e9d/vgg19-dcbb9e9d.pth'))
```

```
Out[5]:
<All keys matched successfully>
```

为了抽取图像的内容特征和样式特征，我们可以选择VGG网络中某些层的输出。一般来说，越靠近输入层的输出越容易抽取图像的细节信息，反之则越容易抽取图像的全局信息。为了避免合成图像过多保留内容图像的细节，我们选择VGG较靠近输出的层，也称内容层，来输出图像的内容特征。我们还从VGG中选择不同层的输出来匹配局部和全局的样式，这些层也叫样式层。在“[使用重复元素的网络（VGG）](#)”一节中我们曾介绍过，VGG网络使用了5个卷积块。实验中，我们选择第四卷积块的最后一个卷积层作为内容层，以及每个卷积块的第一个卷积层作为样式层。这些层的索引可以通过打印`pretrained_net`实例来获取。

```
In [6]:

style_layers, content_layers = [0, 5, 10, 19, 28], [25]
```

在抽取特征时，我们只需要用到VGG从输入层到最靠近输出层的内容层或样式层之间的所有层。下面构建一个新的网络`net`，它只保留需要用到VGG的所有层。我们将使用`net`来抽取特征。

```
In [7]:

net_list = []
for i in range(max(content_layers + style_layers) + 1):
    net_list.append(pretrained_net.features[i])
net = torch.nn.Sequential(*net_list)
```

给定输入`X`，如果简单调用前向计算`net(X)`，只能获得最后一层的输出。由于我们还需要中间层的输出，因此这里我们逐层计算，并保留内容层和样式层的输出。

```
In [8]:

def extract_features(X, content_layers, style_layers):
    contents = []
    styles = []
    for i in range(len(net)):
        X = net[i](X)
        if i in style_layers:
            styles.append(X)
        if i in content_layers:
            contents.append(X)
    return contents, styles
```

下面定义两个函数，其中`get_contents`函数对内容图像抽取内容特征，而`get_styles`函数则对样式图像抽取样式特征。因为在训练时无须改变预训练的VGG的模型参数，所以我们可以训练开始之前就提取出内容图像的内容特征，以及样式图像的样式特征。由于合成图像是样式迁移所需迭代的模型参数，我们只能在训练过程中通过调用`extract_features`函数来抽取合成图像的内容特征和样式特征。



In [9]:

```
def get_contents(image_shape, device):
    content_X = preprocess(content_img, image_shape).to(device)
    contents_Y, _ = extract_features(content_X, content_layers, style_layers)
    return content_X, contents_Y

def get_styles(image_shape, device):
    style_X = preprocess(style_img, image_shape).to(device)
    _, styles_Y = extract_features(style_X, content_layers, style_layers)
    return style_X, styles_Y
```

## 定义损失函数

下面我们来描述样式迁移的损失函数。它由内容损失、样式损失和总变差损失3部分组成。

### 内容损失

与线性回归中的损失函数类似，内容损失通过平方误差函数衡量合成图像与内容图像在内容特征上的差异。平方误差函数的两个输入均为extract\_features函数计算所得到的内容层的输出。

In [10]:

```
def content_loss(Y_hat, Y):
    return F.mse_loss(Y_hat, Y)
```

### 样式损失

样式损失也一样通过平方误差函数衡量合成图像与样式图像在样式上的差异。为了表达样式层输出的样式，我们先通过extract\_features函数计算样式层的输出。假设该输出的样本数为1，通道数为 $c$ ，高和宽分别为 $h$ 和 $w$ ，我们可以把输出变换成 $c$ 行 $hw$ 列的矩阵 $\boldsymbol{X}$ 。矩阵 $\boldsymbol{X}$ 可以看作是由 $c$ 个长度为 $hw$ 的向量 $\boldsymbol{x}_1, \dots, \boldsymbol{x}_c$ 组成的。其中向量 $\boldsymbol{x}_i$ 代表了通道 $i$ 上的样式特征。这些向量的格拉姆矩阵（Gram matrix） $\boldsymbol{X}\boldsymbol{X}^T \in \mathbb{R}^{c \times c}$ 中 $i$ 行 $j$ 列的元素 $\boldsymbol{x}_{ij}$ 即向量 $\boldsymbol{x}_i$ 与 $\boldsymbol{x}_j$ 的内积，它表达了通道 $i$ 和通道 $j$ 上样式特征的相关性。我们用这样的格拉姆矩阵表达样式层输出的样式。需要注意的是，当 $hw$ 的值较大时，格拉姆矩阵中的元素容易出现较大的值。此外，格拉姆矩阵的高和宽皆为通道数 $c$ 。为了让样式损失不受这些值的大小影响，下面定义的gram函数将格拉姆矩阵除以了矩阵中元素的个数，即 $chw$ 。

In [11]:

```
def gram(X):
    num_channels, n = X.shape[1], X.shape[2] * X.shape[3]
    X = X.view(num_channels, n)
    return torch.matmul(X, X.t()) / (num_channels * n)
```

自然地，样式损失的平方误差函数的两个格拉姆矩阵输入分别基于合成图像与样式图像的样式层输出。这里假设基于样式图像的格拉姆矩阵gram\_Y已经预先计算好了。

In [12]:

```
def style_loss(Y_hat, gram_Y):
    return F.mse_loss(gram(Y_hat), gram_Y)
```

### 总变差损失

有时候，我们学到的合成图像里面有大量高频噪点，即有特别亮或者特别暗的颗粒像素。一种常用的降噪方法是总变差降噪（total variation denoising）。假设 $\boldsymbol{x}_{i,j}$ 表示坐标为 $(i, j)$ 的像素值，降低总变差损失

$$\sum_{i,j} |\boldsymbol{x}_{i,j} - \boldsymbol{x}_{i+1,j}| + |\boldsymbol{x}_{i,j} - \boldsymbol{x}_{i,j+1}|$$

能够尽可能使邻近的像素值相似。

In [13]:

```
def tv_loss(Y_hat):
    return 0.5 * (F.l1_loss(Y_hat[:, :, 1:, :], Y_hat[:, :, :-1, :]) +
                  F.l1_loss(Y_hat[:, :, :, 1:], Y_hat[:, :, :, :-1]))
```

### 损失函数

样式迁移的损失函数即内容损失、样式损失和总变差损失的加权和。通过调节这些权值超参数，我们可以权衡合成图像在保留内容、迁移样式以及降噪三方面的相对重要性。

In [14]:

```
content_weight, style_weight, tv_weight = 1, 1e3, 10

def compute_loss(X, contents_Y_hat, styles_Y_hat, contents_Y, styles_Y_gram):
    # 分别计算内容损失、样式损失和总变差损失
    contents_l = [content_loss(Y_hat, Y) * content_weight for Y_hat, Y in zip(
        contents_Y_hat, contents_Y)]
    styles_l = [style_loss(Y_hat, Y) * style_weight for Y_hat, Y in zip(
        styles_Y_hat, styles_Y_gram)]
    tv_l = tv_loss(X) * tv_weight
    # 对所有损失求和
    l = sum(styles_l) + sum(contents_l) + tv_l
    return contents_l, styles_l, tv_l, l
```

## 创建和初始化合成图像

在样式迁移中，合成图像是唯一需要更新的变量。因此，我们可以定义一个简单的模型GeneratedImage，并将合成图像视为模型参数。模型的前向计算只需返回模型参数即可。

In [15]:

```
class GeneratedImage(torch.nn.Module):
    def __init__(self, img_shape):
        super(GeneratedImage, self).__init__()
        self.weight = torch.nn.Parameter(torch.rand(*img_shape))

    def forward(self):
        return self.weight
```

下面，我们定义get\_inits函数。该函数创建了合成图像的模型实例，并将其初始化为图像X。样式图像在各个样式层的格拉姆矩阵styles\_Y\_gram将在训练前预先计算好。

In [16]:

```
def get_inits(X, device, lr, styles_Y):
    gen_img = GeneratedImage(X.shape).to(device)
    gen_img.weight.data = X.data
    optimizer = torch.optim.Adam(gen_img.parameters(), lr=lr)
    styles_Y_gram = [gram(Y) for Y in styles_Y]
    return gen_img(), styles_Y_gram, optimizer
```

## 训练

在训练模型时，我们不断抽取合成图像的内容特征和样式特征，并计算损失函数。

In [17]:

```
def train(X, contents_Y, styles_Y, device, lr, max_epochs, lr_decay_epoch):
    print("training on ", device)
    X, styles_Y_gram, optimizer = get_inits(X, device, lr, styles_Y)
    scheduler = torch.optim.lr_scheduler.StepLR(optimizer, lr_decay_epoch, gamma=0.1)
    for i in range(max_epochs):
        start = time.time()

        contents_Y_hat, styles_Y_hat = extract_features(
            X, content_layers, style_layers)
        contents_l, styles_l, tv_l, l = compute_loss(
            X, contents_Y_hat, styles_Y_hat, contents_Y, styles_Y_gram)

        optimizer.zero_grad()
        l.backward(retain_graph = True)
        optimizer.step()
        scheduler.step()

        if i % 50 == 0 and i != 0:
            print('epoch %3d, content loss %.2f, style loss %.2f, '
                  'TV loss %.2f, %.2f sec'
                  % (i, sum(contents_l).item(), sum(styles_l).item(), tv_l.item(),
                     time.time() - start))
    return X.detach()
```

下面我们开始训练模型。首先将内容图像和样式图像的高和宽分别调整为150和225像素。合成图像将由内容图像来初始化。

In [18]:

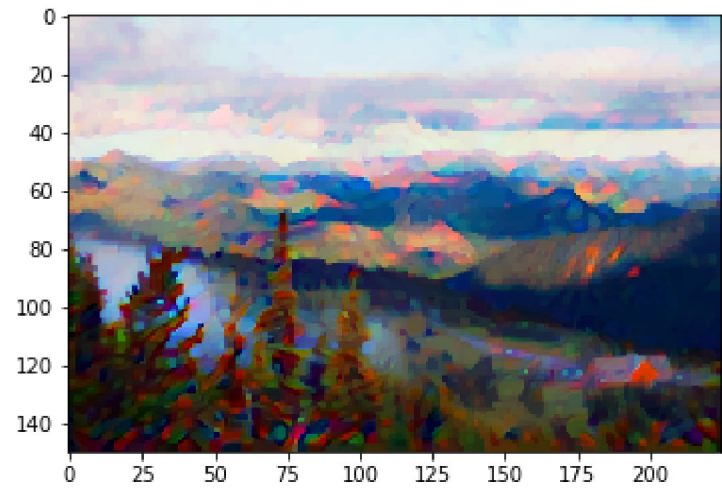
```
image_shape = (150, 225)
net = net.to(device)
content_X, contents_Y = get_contents(image_shape, device)
style_X, styles_Y = get_styles(image_shape, device)
output = train(content_X, contents_Y, styles_Y, device, 0.01, 500, 200)
```

```
training on cpu
epoch  50, content loss 0.24, style loss 1.11, TV loss 1.33, 1.61 sec
epoch 100, content loss 0.24, style loss 0.81, TV loss 1.20, 1.52 sec
epoch 150, content loss 0.24, style loss 0.73, TV loss 1.12, 1.66 sec
epoch 200, content loss 0.23, style loss 0.68, TV loss 1.06, 1.59 sec
epoch 250, content loss 0.23, style loss 0.68, TV loss 1.05, 1.61 sec
epoch 300, content loss 0.23, style loss 0.67, TV loss 1.04, 1.49 sec
epoch 350, content loss 0.23, style loss 0.67, TV loss 1.04, 1.43 sec
epoch 400, content loss 0.23, style loss 0.67, TV loss 1.03, 1.42 sec
epoch 450, content loss 0.23, style loss 0.67, TV loss 1.03, 1.30 sec
```

下面我们将训练好的合成图像保存起来。可以看到图9.14中的合成图像保留了内容图像的风景和物体，并同时迁移了样式图像的色彩。因为图像尺寸较小，所以细节上依然比较模糊。

In [19]:

```
plt.imshow(postprocess(output));
```



为了得到更加清晰的合成图像，下面我们在更大的**300 × 450**尺寸上训练。我们将图9.14的高和宽放大2倍，以初始化更大尺寸的合成图像。

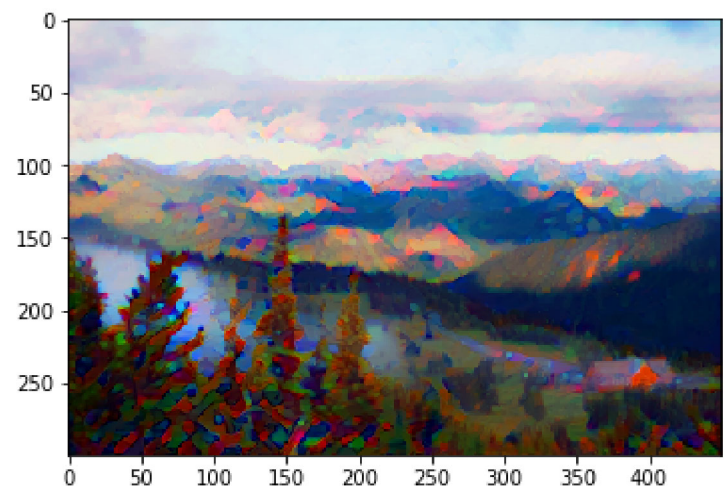
In [21]:

```
image_shape = (300, 450)
_, content_Y = get_contents(image_shape, device)
_, style_Y = get_styles(image_shape, device)
X = preprocess(postprocess(output), image_shape).to(device)
big_output = train(X, content_Y, style_Y, device, 0.01, 100, 200)

training on cpu
epoch  50, content loss 0.34, style loss 0.62, TV loss 0.79, 1.70 sec
```



```
In [22]:  
  
plt.imshow(postprocess(big_output));
```



可以看到，由于图像尺寸更大，每一次迭代需要花费更多的时间。从训练得到的图9.15中可以看到，此时的合成图像因为尺寸更大，所以保留了更多的细节。合成图像里面不仅有大块的类似样式图像的油画色彩块，色彩块中甚至出现了细微的纹理。



### 小结

- 样式迁移常用的损失函数由3部分组成：内容损失使合成图像与内容图像在内容特征上接近，样式损失令合成图像与样式图像在样式特征上接近，而总变差损失则有助于减少合成图像中的噪点。
- 可以通过预训练的卷积神经网络来抽取图像的特征，并通过最小化损失函数来不断更新合成图像。
- 用格拉姆矩阵表达样式层输出的样式。

### 练习

- 选择不同的内容和样式层，输出有什么变化？
- 调整损失函数中的权值超参数，输出是否保留更多内容或减少更多噪点？
- 替换实验中的内容图像和样式图像，你能创作出更有趣的合成图像吗？

### 参考文献

[1] Gatys, L. A., Ecker, A. S., & Bethge, M. (2016). Image style transfer using convolutional neural networks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (pp. 2414-2423).