第5章

迁移手拉手

深度学习已经在各个领域展现出了惊人的能力，但是在实际应用中，我们经常会遇到数据量不足、训练时间过长等问题。迁移学习（Transfer Learning）作为一种解决这些问题的方法，已经在深度学习领域受到了广泛的关注。

风格化迁移，也被称为图像风格迁移，是一种计算机视觉技术，可以将一张图像的艺术风格应用到另一张图像上，从而创造出新的图像。这项技术使用深度学习算法，通过对两张图像的内容和风格进行分析，生成一个新的图像，使其保留原始图像的内容，但以另一张图像的艺术风格呈现。

本章将对迁移学习与风格迁移进行介绍。

5.1 迁移学习

为什么需要迁移学习？原因如下。

（1）大数据与少标注的矛盾。虽然有大量的数据，但往往都是没有标注的，无法训练机器学习模型。人工进行数据标注太耗时。

（2）大数据与弱计算的矛盾。普通人无法拥有庞大的数据量与计算资源。因此需要借助于模型的迁移。

（3）普适化模型与个性化需求的矛盾。即使是在同一个任务上，一个模型也往往难以满足每个人的个性化需求，比如特定的隐私设置。这就需要在不同人之间做模型的适配。

（4）特定应用（如冷启动）的需求。

因此，迁移学习的基本问题主要有以下3个。

- How to transfer：如何进行迁移学习（设计迁移方法）。
- What to transfer：给定一个目标领域，如何找到相对应的源领域，然后进行迁移（源领域选择）。
- When to transfer：什么时候可以进行迁移，什么时候不可以进行迁移（避免负迁移）。

5.1.1　迁移学习的概念

迁移学习是指将已经在一个任务上学习到的知识或模型应用到另一个任务中的过程。其基本原理是通过利用源领域(Source Domain)上学习到的知识,来帮助目标领域(Target Domain)上的学习任务。

其中,任务由函数和目标学习结果组成,是学习的结果;源域是指已有知识的域;目标域为要进行学习的域。

按不同的分类方法,迁移学习有不同的学习方法。

1. 按特征空间分类

按特征空间分类,迁移学习可分为以下两种。

(1) 同构迁移学习(Homogeneous TL):源域和目标域的特征空间相同,转存失败重新上传。

(2) 异构迁移学习(Heterogeneous TL):源域和目标域的特征空间不同,转存失败重新上传取消。

2. 按迁移情景分类

按迁移情景分类,迁移学习可分为以下三种。

(1) 归纳式迁移学习(Inductive TL):源域和目标域的学习任务不同。

(2) 直推式迁移学习(Transductive TL):源域和目标域不同,学习任务相同。

(3) 无监督迁移学习(Unsupervised TL):源域和目标域均没有标签。

3. 按迁移方法分类

按迁移方法分类,迁移学习可分为以下三种。

(1) 基于样本的迁移(Instance-based TL):通过权重重用源域和目标域的样例进行迁移。

基于样本的迁移学习方法是根据一定的权重生成规则,对数据样本进行重用来进行迁移学习。图 5-1 形象地表示了基于样本迁移方法的源域中存在不同种类的动物,如狗、鸟、猫等,目标域只有狗这一种类别。在迁移时,为了最大限度地和目标域相似,我们可以人为地提高源域中属于狗这个类别的样本权重。

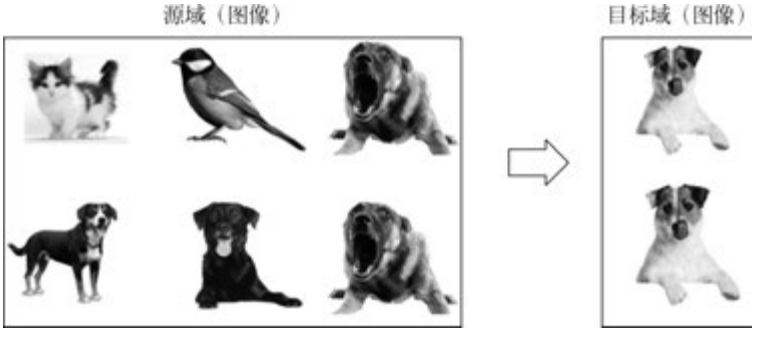

图 5-1　基于样本迁移学习方法的效果

(2) 基于特征的迁移(Feature-based TL):将源域和目标域的特征变换到相同空间。

基于特征的迁移学习方法是指将通过特征变换的方式互相迁移,来减少源域和目标域

之间的差距；或者将源域和目标域的数据特征变换到同一特征空间中，然后利用传统的机器学习方法进行分类识别。根据特征的同构和异构性，又可以分为同构和异构迁移学习。图 5-2 很形象地表示了两种基于特征的迁移学习方法。

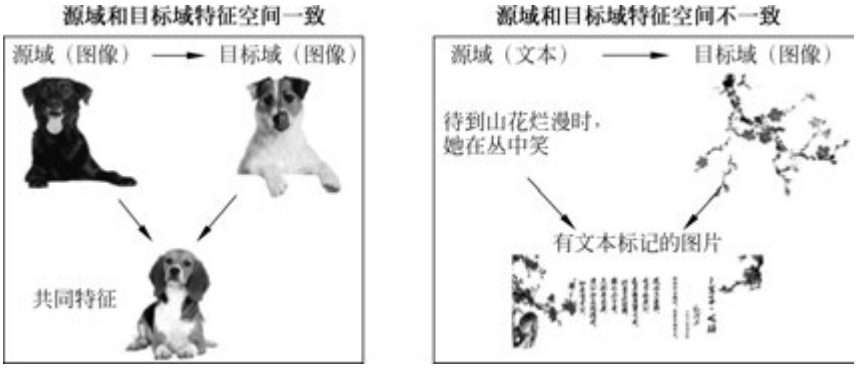

图 5-2　基于特征的迁移学习方法的效果

（3）基于模型的迁移（model-based TL）[也称基于参数的迁移（Parameter-based TL）]：利用源域和目标域的参数共享模型。

基于模型的迁移学习方法是指从源域和目标域中找到它们之间共享的参数信息，以实现迁移的方法。这种迁移方式要求的假设条件是：源域中的数据与目标域中的数据可以共享一些模型的参数。图 5-3 形象地表示了基于模型的迁移学习方法的基本思想。

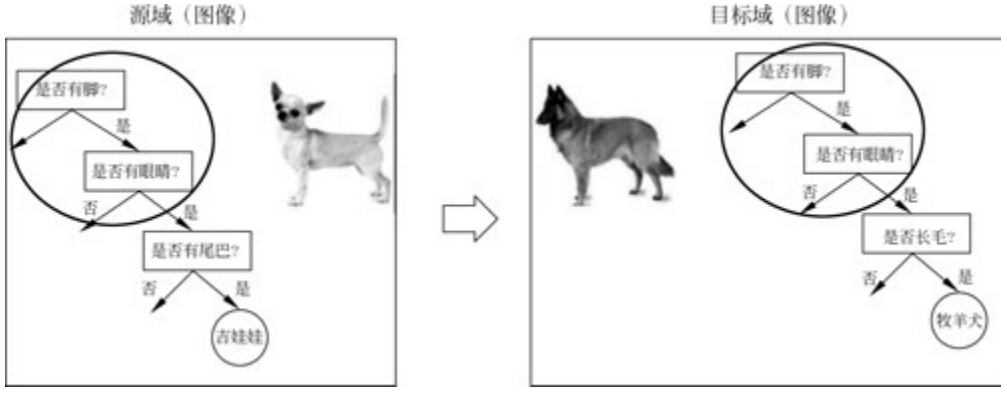

图 5-3　基于模型的迁移学习方法的基本思想

（4）基于关系的迁移（Relation-based TL）：利用源域中的逻辑网络关系进行迁移。

基于关系的迁移学习方法与上述三种方法具有截然不同的思路。这种方法比较关注源域和目标域的样本之间的关系。图 5-4 形象地表示了不同领域之间相似的关系。

各迁移学习方法可用图 5-5 所示的思维导图来展示它们之间的关系。

5.1.2　迁移学习的核心与方法

迁移学习的核心是：找到源域和目标域之间的相似性，并加以合理利用。这种相似性非常普遍。例如，不同人的身体构造是相似的，自行车和摩托车的骑行方式是相似的，国际

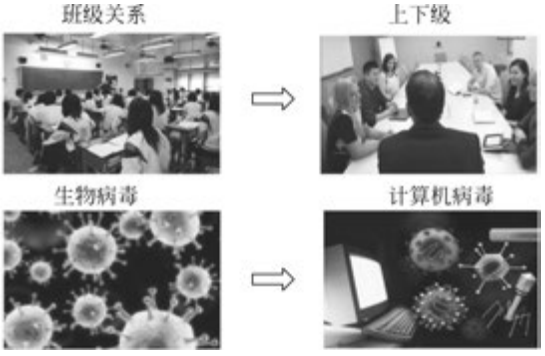

图 5-4　基于关系的迁移学习方法的效果

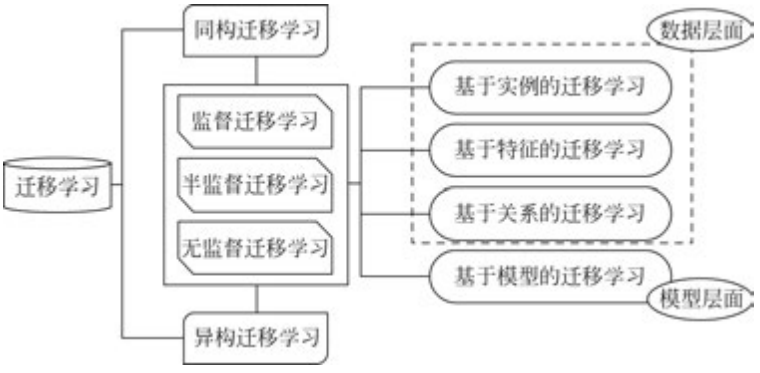

图 5-5　各迁移学习的关系图

象棋和中国象棋是相似的，羽毛球和网球的打球方式是相似的。这种相似性也可以理解为不变量。以不变应万变，才能立于不败之地。

因此，迁移学习的核心方法如下。

1. 特征重用

特征重用(Feature Reuse)是迁移学习一种简单但有效的方法，通过直接使用源任务模型的特征提取层，将其应用到目标任务中进行特征提取，再在目标任务的数据上训练新的分类器或回归器。

【例 5-1】 特征重用方法实例演示。

```
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.applications import VGG16

#加载预训练的 VGG16 模型,不包括顶层分类器
base_model = VGG16(weights = 'imagenet', include_top = False, input_shape = (224, 224, 3))
#冻结预训练模型的层
for layer in base_model.layers:
    layer.trainable = False
#构建新的分类器
model = models.Sequential([
    base_model,
    layers.Flatten(),
```

```
    layers.Dense(256, activation = 'relu'),
    layers.Dropout(0.5),
    layers.Dense(10, activation = 'softmax')
])
#编译模型
model.compile(optimizer = 'adam', loss = 'categorical_crossentropy', metrics = ['accuracy'])
#加载并预处理 CIFAR-10 数据集
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar10.load_data()
x_train = tf.image.resize(x_train, (224, 224)).numpy() / 255.0
x_test = tf.image.resize(x_test, (224, 224)).numpy() / 255.0
y_train = tf.keras.utils.to_categorical(y_train, 10)
y_test = tf.keras.utils.to_categorical(y_test, 10)
#训练模型
model.fit(x_train, y_train, epochs = 10, validation_data = (x_test, y_test), batch_size = 32)
#评估模型
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f'测试准确率: {test_acc}')
```

运行程序,输出如下:

```
测试准确率:0.87241
```

2. 微调

微调(Fine-Tuning)是迁移学习的一种常用方法,通过在目标任务的数据上继续训练预训练模型的部分或全部层,从而适应目标任务的特性。

【例 5-2】 迁移学习之微调方法学实例演示。

```
#解冻部分预训练模型的层
for layer in base_model.layers[-4:]:
    layer.trainable = True
#重新编译模型(使用较小的学习率)
model.compile(optimizer = tf.keras.optimizers.Adam(learning_rate = 1e-5), loss =
'categorical_crossentropy', metrics = ['accuracy'])
#继续训练模型
model.fit(x_train, y_train, epochs = 5, validation_data = (x_test, y_test), batch_size = 32)
#评估模型
test_loss, test_acc = model.evaluate(x_test, y_test)
print(f'微调后的测试准确率: {test_acc}')
```

运行程序,输出如下:

```
微调后的测试准确率: 0.89045
```

3. 领域适应

领域适应(Domain Adaptation)是迁移学习中的一种方法,通过调整源域模型使其能够更好地适应目标域的数据分布,从而提高在目标域的预测性能。常见的领域适应方法包括对抗训练(Adversarial Training)和子空间对齐(Subspace Alignment)等。

【例 5-3】 领域适应方法实例演示。

```
from tensorflow.keras.datasets import mnist, usps
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Dense, Input
```

```
#加载 MNIST 和 USPS 数据集
(mnist_train_images, mnist_train_labels), (mnist_test_images, mnist_test_labels) = mnist.
load_data()
(usps_train_images, usps_train_labels), (usps_test_images, usps_test_labels) = usps.load_
data()
#数据预处理
mnist_train_images = mnist_train_images.reshape(-1, 28 * 28).astype('float32') / 255
mnist_test_images = mnist_test_images.reshape(-1, 28 * 28).astype('float32') / 255
usps_train_images = usps_train_images.reshape(-1, 28 * 28).astype('float32') / 255
usps_test_images = usps_test_images.reshape(-1, 28 * 28).astype('float32') / 255
mnist_train_labels = tf.keras.utils.to_categorical(mnist_train_labels, 10)
mnist_test_labels = tf.keras.utils.to_categorical(mnist_test_labels, 10)
usps_train_labels = tf.keras.utils.to_categorical(usps_train_labels, 10)
usps_test_labels = tf.keras.utils.to_categorical(usps_test_labels, 10)
#定义源域模型
input_tensor = Input(shape = (28 * 28,))
x = Dense(256, activation = 'relu')(input_tensor)
x = Dense(256, activation = 'relu')(x)
output_tensor = Dense(10, activation = 'softmax')(x)
source_model = Model(inputs = input_tensor, outputs = output_tensor)
source_model.compile(optimizer = 'adam', loss = 'categorical_crossentropy', metrics = ['accuracy'])
#在 MNIST 数据集上训练源域模型
source_model.fit(mnist_train_images, mnist_train_labels, epochs = 10, batch_size = 128,
validation_data = (mnist_test_images, mnist_test_labels))
#定义领域适应模型
feature_extractor = Model(inputs = source_model.input, outputs = source_model.layers[-2].
output)
target_input = Input(shape = (28 * 28,))
target_features = feature_extractor(target_input)
target_output = Dense(10, activation = 'softmax')(target_features)
domain_adapt_model = Model(inputs = target_input, outputs = target_output)
domain_adapt_model.compile(optimizer = 'adam', loss = 'categorical_crossentropy', metrics =
['accuracy'])
#在 USPS 数据集上微调领域适应模型
domain_adapt_model.fit(usps_train_images, usps_train_labels, epochs = 10, batch_size = 128,
validation_data = (usps_test_images, usps_test_labels))
#评估领域适应模型
test_loss, test_acc = domain_adapt_model.evaluate(usps_test_images, usps_test_labels)
print(f'领域适应模型在 USPS 测试集上的准确率: {test_acc}')
```

运行程序,输出如下:

```
领域适应模型在 USPS 测试集上的准确率:0.89364
```

5.2 风格迁移

除 DeepDream(深度梦境)外,深度学习驱动图像修改的另一项重大进展是神经风格迁移(neural style transfer)。自首次提出以来,神经风格迁移算法已做许多改进,并衍生出许多变体,还成功转化成许多智能手机图片应用。

5.2.1 风格迁移的定义

风格迁移又称风格转换，只需给定原始图片，并选择艺术家的风格图片，就能把原始图片转换成具有相应艺术家风格的图片，并同时保留目标图像的内容，图 5-6 给出了一组实例。

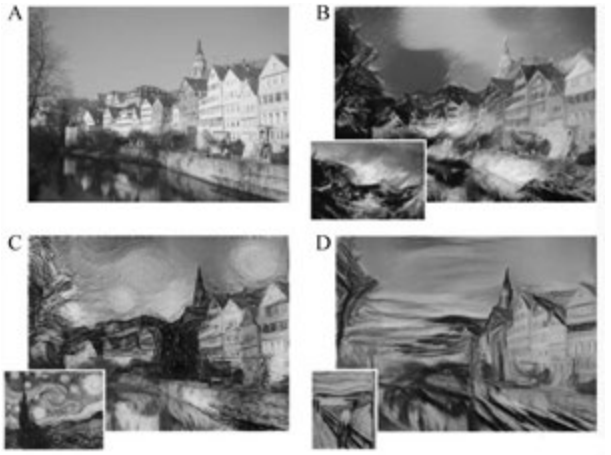

图 5-6 一组风格迁移实例

5.2.2 风格迁移的方法

图 5-7 用简单的实例阐述了基于卷积神经网络的风格迁移方法。首先，初始化合成图像，例如将其初始化为内容图像，该合成图像是风格迁移过程中唯一需要更新的变量，即风格迁移所需迭代的模型参数。然后，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。这个深度卷积神经网络凭借多个层逐级抽取图像的特征，可以选择其中某些层的输出作为内容特征或风格特征。以图 5-7 为例，此处选取的预训练的神经网络含有 3 个卷积层，其中第二层输出内容特征，第一层和第三层输出风格特征。

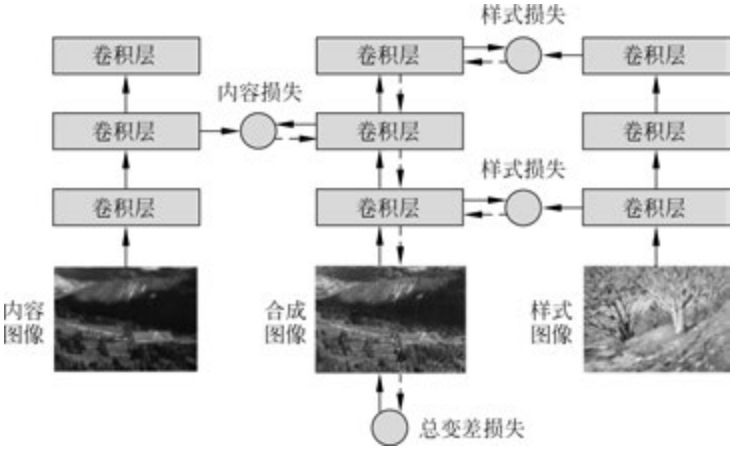

图 5-7 基于卷积神经网络的风格迁移方法

接着，通过前向传播（实线箭头方向）计算风格迁移的损失函数，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。风格迁移常用的损失函数由以下 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；
- 风格损失使合成图像与风格图像在风格特征上接近；
- 全变分损失则有助于减少合成图像中的噪点。

最后，当模型训练结束时，输出风格迁移的模型参数，即得到最终的合成图像。

5.2.3　风格迁移的实现

本节通过一个实例来演示风格迁移过程。

1. 输入图像

首先，读取内容和风格图像，从输出的图像坐标轴可以看出，它们的尺寸并不一样。

```
%matplotlib inline
import torch
import torchvision
from torch import nn
from d2l import torch as d2l

d2l.set_figsize()
content_img = d2l.Image.open('img3.jpg')
d2l.plt.imshow(content_img);                    #效果如图 5-8 所示
style_img = d2l.Image.open('img4.jpg')
d2l.plt.imshow(style_img);                      #效果如图 5-9 所示
```

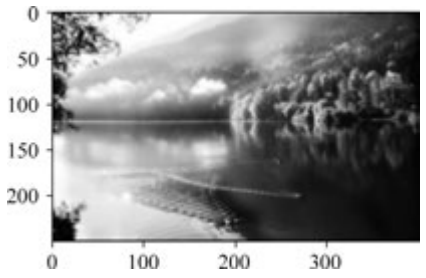

图 5-8　原始输入图像

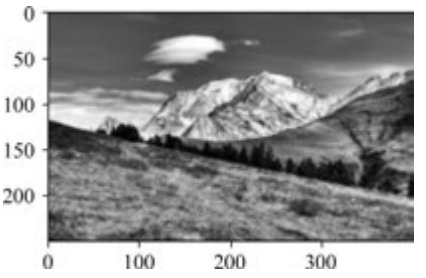

图 5-9　风格图像

2. 预处理与后处理

接下来，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 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 和 1 之间，对小于 0 和大于 1 的值分别取 0 和 1。

提示：torchvision.transforms 模块有大量现成的转换方法，不过需要注意的是，有的方法输入的是 PIL 图像，如 Resize；有的方法输入的是 tensor，如 Normalize；还有的是用于二者转换，如 ToTensor 将 PIL 图像转换成 tensor。

```
rgb_mean = torch.tensor([0.485, 0.456, 0.406])
```

```
rgb_std = torch.tensor([0.229, 0.224, 0.225])

def preprocess(img, image_shape):
    transforms = torchvision.transforms.Compose([
        torchvision.transforms.Resize(image_shape),
        torchvision.transforms.ToTensor(),
        torchvision.transforms.Normalize(mean = rgb_mean, std = rgb_std)])
    return transforms(img).unsqueeze(0)

def postprocess(img):
    img = img[0].to(rgb_std.device)
    img = torch.clamp(img.permute(1, 2, 0) * rgb_std + rgb_mean, 0, 1)
    return torchvision.transforms.ToPILImage()(img.permute(2, 0, 1))
```

3. 抽取图像特征

使用基于 ImageNet 数据集预训练的 VGG19 模型来抽取图像特征。

提示：PyTorch 的 torchvision.models 模块提供了一些常见的预训练好的计算机视觉模型，包括图片分类、语义分割、目标检测、实例分割、人关键点检测和视频分类等。

```
pretrained_net = torchvision.models.vgg19(pretrained = True)
```

为了抽取图像的内容特征和风格特征，可以选择 VGG 网络中某些层的输出。一般来说，越靠近输入层，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，可选择 VGG 靠近输出的层（内容层）来输出图像的内容特征，还可从 VGG 中选择不同层的输出来匹配局部和全局的风格，这些图层也称为风格层。

在该实例中，VGG 网络使用了 5 个卷积块，选择第 4 个卷积层的最后一个卷积层作为内容层，选择每个卷积块的第一个卷积层作为风格层。

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

使用 VGG 层抽取特征时，只需要用到从输入层到最靠近输出层的内容层或风格层之间的所有层。下面代码构建一个新的网络 net，它用来保留用到的所有 VGG 层。

```
net = nn.Sequential( * [pretrained_net.features[i] for i in range(max(content_layers + style_
layers) + 1)])
```

给定输入 X，如果简单地调用前向传播 net(x)，只能获得最后一层的输出。由于还需要中间层的输出，此处需要逐层计算，并保留内容层和风格层的输出。

```
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
```

接下来定义两个函数：函数 get_contents 用于抽取内容图像的内容特征；函数 get_

styles 用于抽取风格图像的风格特征。因为在训练时无须改变预训练的 VGG 模型参数，所以可以在训练开始前提取出内容特征和风格特征。由于合成图像是风格迁移所需迭代的模型参数，因此只能在训练过程中通过调用函数 extract_features 来提取图像的内容特征和风格特征。

```
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
```

4. 定义损失函数

下面来描述风格迁移的损失函数，它由内容损失、风格损失和全变分损失 3 部分组成。

1）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函数 extract_features 计算所得到的内容层的输出。

```
def content_loss(Y_hat, Y):
    #从动态计算梯度的树中分离目标：这是一个规定的值，而不是一个变量
    return torch.square(Y_hat - Y.detach()).mean()
```

2）风格损失

风格损失与内容损失类似，也通过平方误差函数衡量合成图像与风格图像在风格上的差异。为了表达风格层输出的风格，先通过函数 extract_features 计算风格层的输出。假设该输出样本为 1，通道数为 c，高和宽分别为 h 和 w，可以将此输出转换为矩阵 $\boldsymbol{X}$，其有 c 行和 $h \times w$ 列。这个矩阵可以被看作是由 c 个长度为 $h \times w$ 的向量 $\boldsymbol{x}_1, \cdots, \boldsymbol{x}_c$ 组合而成。其中向量 $\boldsymbol{x}_i$ 代表了通道 i 上的风格特征。

在向量的格拉姆矩阵 $\boldsymbol{X}\boldsymbol{X}^{\mathrm{T}} \in \mathbf{R}^{c \times c}$ 中，i 行 j 列的元素 x_{ij} 即向量 $\boldsymbol{x}_i$ 和 $\boldsymbol{x}_j$ 的内积。它表达了通道 i 和通道 j 上风格特征的相关性，用这样的格拉姆矩阵来表达风格层输出的风格。值得注意的是，当 $h \times w$ 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。为了让风格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 $c \times h \times w$）。

```
def gram(X):
    num_channels, n = X.shape[1], X.numel() // X.shape[1]
    X = X.reshape((num_channels, n))
    return torch.matmul(X, X.T) / (num_channels * n)
```

风格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风格图像的风格层输出。此处假设基于风格图像的格拉姆矩阵 gram_Y 已经预先计算好了。

```
def style_loss(Y_hat, gram_Y):
    return torch.square(gram(Y_hat) - gram_Y.detach()).mean()
```

3）全变分损失

有时，在合成的图像中有大量高频噪声，即有特别亮或特别暗的颗粒像素。一种常见的去噪方法是全变分去噪(total variation denoising)：假设 $x_{i,j}$ 表示坐标(i,j)处的像素值，降低全变分损失：

$$\sum_{i,j}(|x_{i,j}-x_{i+1,j}|+|x_{i,j}-x_{i,j+1}|)$$

尽可能使邻近的像素值相似。

```
def tv_loss(Y_hat):
    return 0.5 * (torch.abs(Y_hat[:, :, 1:, :] - Y_hat[:, :, :-1, :]).mean() +
                  torch.abs(Y_hat[:, :, :, 1:] - Y_hat[:, :, :, :-1]).mean())
```

4）损失函数

风格转换的损失函数是内容损失、风格损失和全变分损失的加权和。通过调节这些权重超参数，可以权衡合成图像在保留内容、迁移风格及去噪三方面的相对重要性。

```
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、风格损失和全变分损失
    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(10 * styles_l + contents_l + [tv_l])
    return contents_l, styles_l, tv_l, l
https://inscode.csdn.net/?utm_source=260232576
```

5. 初始化合成图像

在风格迁移中，合成的图像是训练期间唯一需要更新的变量。因此，可以定义一个简单的模型 SynthesizedImage，并将合并成的图像视为模型参数。模型的前向传播只需返回模型参数即可。

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

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

下面代码实现 get_inits 函数定义，get_inits 函数创建了合成图像的模型实例，并将其初始化为图像 X。风格图像在各个风格层的格拉姆矩阵 styles_Y_gram 将在训练前预先计算好。

```
def get_inits(X, device, lr, styles_Y):
    gen_img = SynthesizedImage(X.shape).to(device)
    gen_img.weight.data.copy_(X.data)
    trainer = torch.optim.Adam(gen_img.parameters(), lr=lr)
```

```
    styles_Y_gram = [gram(Y) for Y in styles_Y]
    return gen_img(), styles_Y_gram, trainer
```

6. 模型训练

在进行模型训练风格迁移时，不断提取合成图像的内容特征和风格特征，然后计算损失函数：

```
#定义训练循环
def train(X, contents_Y, styles_Y, device, lr, num_epochs, lr_decay_epoch):
    X, styles_Y_gram, trainer = get_inits(X, device, lr, styles_Y)
    scheduler = torch.optim.lr_scheduler.StepLR(trainer, lr_decay_epoch, 0.8)
    animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                            xlim=[10, num_epochs],
                            legend=['content', 'style', 'TV'],
                            ncols=2, figsize=(7, 2.5))
    for epoch in range(num_epochs):
        trainer.zero_grad()
        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)
        l.backward()
        trainer.step()
        scheduler.step()
        if (epoch + 1) % 10 == 0:
            animator.axes[1].imshow(postprocess(X))
            animator.add(epoch + 1, [float(sum(contents_l)),
                                     float(sum(styles_l)), float(tv_l)])
    return X
```

接着训练模型：先将内容图像和风格图像的高和宽分别调整为 300 和 450 像素，合成图像是用内容图像来实现初始化。

```
device, image_shape = d2l.try_gpu(), (300, 450)
net = net.to(device)
content_X, contents_Y = get_contents(image_shape, device)
_, styles_Y = get_styles(image_shape, device)
output = train(content_X, contents_Y, styles_Y, device, 0.3, 500, 50)
```

运行程序，效果如图 5-10 所示。

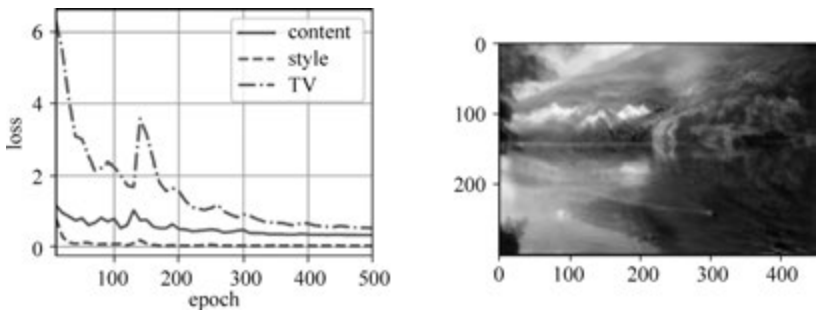

图 5-10　图像风格迁移效果